Habilitação técnica em

Informática

3

Análise e Gerenciamento de Dados

Centro Paula Souza

CENTRO PAULA SOUZA DO GOVERNO DE SÃO PAULO
CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

Centro Paula Souza

Informática

Volume 3

CENTRO PAULA SOUZA

Informática

Análise e gerenciamento de dados

Gustavo Dibbern Piva
Wilson José de Oliveira

FUNDAÇÃO PADRE ANCHIETA
São Paulo
2010

Sumário

Capa: Nathalia Guarienti, aluna da Etec do Centro Paula Souza.
Foto: Eduardo Pozella
Edição: Deise Bitinas

Sumário

Sumário

<<ator>>

Capítulo 1

Introdução ao desenvolvimento de software

- Conceitos iniciais
- Gerações de computadores
- Gerações de linguagem de programação
- Processo de desenvolvimento
- Modelo de ciclo de vida
- Riscos
- Prototipagem
- Levantamento ou especificação de requisitos

A automação, há algum tempo, tornou-se um diferencial competitivo para as organizações. Tal modernização pode ser entendida como o esforço para transformar as tarefas manuais repetitivas em processos independentes, realizados por máquinas. Com isso, a gestão passou por uma verdadeira revolução: erros que antes eram cometidos por falhas de cálculos agora são quase nulos. As empresas não precisam se voltar tanto para atividades contínuas e podem se dedicar mais ao foco do seu negócio. Os benefícios disso se estendem por toda a cadeia de valor, desde compra de materiais, produção e vendas até entrega de produtos para os clientes e pós-venda.

O desenvolvimento de software é uma atividade muito complexa. Sim, porque não existe uma solução pronta para cada cenário. O trabalho é sempre realizado por pessoas, o que torna o sucesso do projeto diretamente relacionado à competência da equipe e à maneira como se trabalha. Outra dificuldade considerável é o fato de, muitas vezes, não ser adotado um processo definido para apoiar essa tarefa.

A tecnologia da informação passou por uma grande evolução nos últimos anos. Com isso, há exigências contínuas e renovadas de aumento na qualificação dos profissionais da área, o que, consequentemente, favorece o seu desenvolvimento. Afinal, esse segmento é composto pela tríade pessoas, gestão e tecnologia.

O emprego desse tripé levou a uma significativa melhoria no desenvolvimento das companhias chamadas inteligentes, aquelas que necessitam de sistemas tolerantes a falhas e capazes de gerar informações críticas para o processo de decisão. Na década de 1960 as empresas trabalhavam com o conceito de processamento centralizado de dados e muitos dos seus recursos eram direcionados ao CPD (Centro de Processamento de Dados). Os sistemas daquela época rodavam de forma mecanizada e em batch (processamento em lote).

Com o passar do tempo, porém, as corporações perceberam a necessidade e a importância de se basear em informações concretas para tomar suas decisões e, assim, aprimorar a gestão dos negócios. Então, abandonaram o padrão do tradicional processamento de dados e passaram a trabalhar com centro de informações. Nesse modelo, já havia integração dos sistemas, mesmo que ainda existissem algumas redundâncias, ou seja, dados duplicados que levavam ao retrabalho.

Atualmente, as organizações vivem a era da Tecnologia da Informação (TI). Agora, os sistemas são todos integrados, possibilitando a otimização dos processos e a diminuição da redundância de dados. Com isso, é possível melhorar muito o dempenho da empresa, pois os processos se tornam mais estruturados, fato que minimiza o retrabalho (**REZENDE**, 2002).

Denis Alcides Rezende é autor de vários livros sobre Tecnologia da Informação, Sistemas de Informação e Engenharia de Software. Atua como consultor em planejamento de Tecnologia da Informação.

1.1. Conceitos iniciais

Com a crescente necessidade das empresas de contar com informações cada vez mais rápidas e confiáveis, foi necessário desenvolver não apenas sistemas, como também novos hardwares. Era preciso ter máquinas que atendessem às especificações de softwares de alto desempenho e de grande disponibilidade.

1.2. Gerações de computadores

Vamos conhecer agora as cinco gerações de equipamentos, a tecnologia empregada em cada uma delas, suas vantagens e desvantagens.

As cinco gerações de máquinas

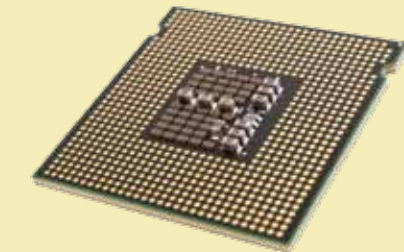
HULTON ARCHIVE/GETTY IMAGES

1ª • 1940 - 1958
Univac I: dez vezes mais rápido e com um décimo do tamanho do Eniac, o modelo anterior

2ª • 1958 - 1964
IBM 7090: as válvulas deram lugar aos transistores e a vida útil dos equipamentos aumentou

CERN/SCIENCE PHOTO LIBRARY/LATINSTOCK

SHEILA TERRY/RUTHERFORD APPLETON

3ª • 1964 - 1971
Com a série 360, da IBM, a velocidade dos equipamentos passou para bilionésimos de segundo

4ª • 1971 - 1987
Chega a era do chip, a lâmina de silício que permite a integração de uma infinidade de circuitos

OLEKSIY MAKSYMENKO/ALAMY/OTHER IMAGES

DIVULGAÇÃO

5ª • 1987
Novas tecnologias revolucionam os conceitos de processamento paralelo e armazenamento de dados

• Primeira (1940-1958)

O primeiro computador digital eletrônico de grande escala foi o Eniac (Electrical Numerical Integrator and Calculator, ou integradora e calculadora numérica elétrica). Começou a ser desenvolvido em 1943, durante a Segunda Guerra Mundial, para auxiliar o Exército dos Estados Unidos nos cálculos de balística. Foi lançado em fevereiro de 1946 pelos cientistas norte-americanos John Presper Eckert e John W. Mauchly, da Electronic Control Company. O Eniac realizava 5 mil operações por segundo, velocidade mil vezes superior à das máquinas da época. No entanto, se comparado com os computadores atuais, o seu poder de processamento seria menor do que o de uma simples calculadora de bolso.

Mas a primeira geração de máquinas teve como marco histórico o lançamento do primeiro computador comercial, o Univac I (Universal Automatic Computer ou computador automático universal), em 1951. Ele possuía cem vezes a capacidade do Eniac, era dez vezes mais rápido e tinha um décimo de seu tamanho. O Univac I tinha como componentes entre 10 mil e 20 mil válvulas eletrônicas, com duração média de oitocentos a mil horas.

O primeiro modelo do Univac foi construído pela empresa Eckert-Mauchly Computer Corporation, adquirida pela Remington Rand pouco depois. Hoje, os direitos sobre o nome Univac pertencem à Unisys, que aponta a Força Aérea Americana, o Exército e a Comissão de Energia Atômica norte-americanos como seus primeiros clientes. Inicialmente, o Univac era usado para executar funções no Escritório de Censo dos Estados Unidos. Além de órgãos governamentais, eram usuárias do computador empresas como a General Electric, a Metropolitan Life e a Du Pont. Naquela época, cada uma das 46 unidades fabricadas do Univac custava US$ 1 milhão. Em 1953, surgiu o IBM 701 e, em 1954, o IBM 650. Ambos tiveram muito sucesso de vendas para a época, chegando a 2 mil unidades em cinco anos.

TIM RIDLEY/GETTY IMAGES

• Segunda (1958-1964)

A fabricação dos computadores foi alterada e as válvulas deram lugar aos **transistores**, cuja vida útil era bem maior (90 mil horas). Com isso, as máquinas ficaram mais rápidas e menores. Pertence a essa segunda geração o IBM 7090, que possuía 40 mil transistores e 1,2 milhão de núcleos magnéticos. Ocupava uma área de 40 m^2, contra a de 180 m^2 do Eniac.

TIM RIDLEY/GETTY IMAGES

• Terceira (1964-1971)

Em 1964, a IBM anunciou o lançamento da série 360 com **circuito integrado**. Os circuitos impressos são milimétricos e podem conter transistores, resistências e condensadores. Para se ter uma ideia, 50 mil desses conjuntos cabem em um dedal. A novidade permitiu um grande aumento na velocidade de computação de dados, passando de milionésimos de segundo para bilionésimos de segundo. Assim, um programa que era executado em uma hora, no modelo de computador de 1951, levava entre três e quatro segundos para rodar nos equipamentos da terceira geração.

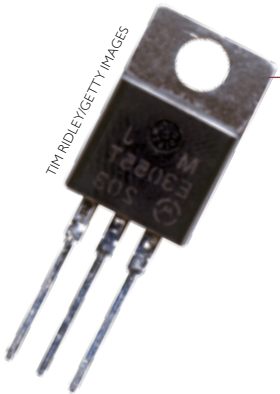
• Quarta (1971-1987)

A integração de circuitos teve grandes incrementos, com crescimento de mil vezes a cada dez anos. Foram introduzidas várias escalas de incorporação, definidas pelo número de conjuntos que podem ser colocados em uma única lâmina miniaturizada feita de silício, o famoso chip. Assim, foram surgindo tecnologias consecutivas: em 1970, LSI (Large Scale Integration – integração em larga escala); em 1975, VLSI (Very Large Scale Integration – integração em muito larga escala); e, em 1980, ULSI (Ultra Large Scale Integration – integração em ultralarga escala).

• Quinta (1987- primeira década do século 21)

A geração de equipamentos em uso na primeira década do século 21 surgiu em 1987, com o uso de novas tecnologias, principalmente relacionadas a dispositivos ópticos e a telecomunicações. Houve aumento de processamento paralelo, diversidade de formato, incremento da capacidade computacional e de armazenamento, assim como difusão do processamento distribuído. Além disso, delineou-se a tendência de convergência de computadores e aparelhos de comunicação, o que facilitou a interoperabilidade e a universalização da operação dos sistemas, assim como a sua normatização.

Vantagens e desvantagens das 5 gerações de computadores

Geração	Componente eletrônico	Vantagens	Desvantagens
1ª geração 1940-1958	válvulas eletrônicas	• únicos componentes eletrônicos disponíveis	• grande dimensão • produzem muito calor • necessitam de ar condicionado
2ª geração 1958-1964	transistores	• menor dimensão • produzem menos calor • mais rápidos	• ainda necessitam de constante manutenção • necessitam de ar condicionado
3ª geração 1964-1971	circuitos integrados	• ainda menor dimensão • menor produção de calor • menor consumo de energia • ainda mais rápidos	• inicialmente com muitos problemas de fábrica
4ª geração 1971-1987	circuitos integrados larga escala	• não é necessário ar condicionado • conservação mínima • alta densidade de componentes	• existem ainda computadores com menos potência em relação a computadores de outras gerações
5ª geração 1987-atual	transdutores e circuitos em paralelo	• maior densidade de componentes • reduzido tamanho • auto-regeneração • grande fiabilidade e velocidade • multiprocessamento	• maior complexidade • ainda muito caros

1.3. Gerações de linguagem de programação

É importante conhecer as gerações de linguagens de programação, para entender bem o contexto atual (**SWEBOK**, 2004).

O Swebok (Software Engineering Body of Knowledge, traduzido por Áreas do Conhecimento da Engenharia de Software) é uma iniciativa da Sociedade da Computação do Instituto de Engenharia Elétrica e Eletrônica (IEEE Computer Society), com o propósito de criar um consenso sobre as áreas de conhecimento da engenharia de software. Publicado em 2004, contou com a participação da indústria internacional, de sociedades de profissionais, da academia e de diversos autores consagrados.

Primeira (1GL)

A primeira geração de programação utiliza apenas linguagem de máquina, ou seja, o sistema binário de 0 (zero) e 1 (um) para o desenvolvimento dos softwares. Sua desvantagem é ser pouco intuitiva, pois não utiliza linguagens mais sofisticadas que permitem a portabilidade do programa, isto é, o código utilizado acaba restrito a um único tipo de hardware e à arquitetura utilizada.

Segunda (2GL)

A linguagem de programação chamada Assembly representa a segunda geração. Mais próxima do ser humano do que da máquina (como acontecia na 1GL), cada Assembly ainda é bastante associada à arquitetura do computador, fazendo com que a 2GL também seja pouco portável entre ambientes.

Terceira (3GL)

A terceira geração das linguagens de programação está muito próxima do ser humano, pois é facilmente entendida por uma pessoa com pouco – ou nenhum – conhecimento de informática. Isso porque é similar à comunicação do dia a dia. Essa geração é representada pelas linguagens Cobol, Fortran, Algol, Basic, C, C++, entre outras. Para mais informações sobre a 3GL veja quadro abaixo.

Quarta (4GL)

A linguagem SQL (Structured Query Language, ou Linguagem de Consulta Estruturada) tornou-se tão popular que passou a representar toda a quarta geração. Ainda hoje, é bastante utilizada como linguagem de manipulação e consulta de banco de dados, como o SQL-Server da Microsoft, Oracle Database da Oracle e MySQL da Sun.

A principal característica das linguagens de quarta geração é descrever o que deve ser feito, permitindo ao programador visar um resultado imediato. São consideradas capazes de, por si sós, gerar códigos, ou seja, os RADs (Rapid Application Development, ou Desenvolvimento Rápido de Aplicação), com os quais podem ser criadas aplicações, mesmo sem se especializar na linguagem. Também nesse grupo estão as linguagens orientadas a objetos.

Quinta (5GL)

A quinta geração ainda está pouco desenvolvida e engloba as linguagens para inteligência artificial. Sua maior representante é a LISP, nome originado da expressão LISt Processing (Processamento em Lista), já que a lista é a sua estrutura de dados fundamental.

Para desenvolver um software pode-se utilizar uma geração de linguagem ou um conjunto delas. Entretanto, o melhor resultado no projeto final é obtido quando se vence o desafio de adequar a programação aos sistemas de informação de diversas áreas do conhecimento.

Saiba mais sobre as 3GL

Cobol, sigla para COmmon Business Oriented Language (Linguagem Orientada aos Negócios): usada em sistemas comerciais, financeiros e administrativos para empresas e governos. Foi criada em 1959, durante o CODASYL (Conference on Data Systems Language, a Conferência de Linguagem de Sistemas de Dados), um dos três comitês propostos em uma reunião no Pentágono, organizado por Charles Phillips, do Departamento de Defesa dos Estados Unidos. As fontes de inspiração são as linguagens FLOW-MATIC, inventada por Grace Hopper, e COMTRAN, da IBM, inventada por Bob Bemer.

Fortran, acrônimo para a expressão *IBM Mathematical FORmula TRANslation System* (Sistema de Tradução de Fórmula Matemática da IBM): família desenvolvida a partir dos anos 1950. Usada, principalmente, em Ciência da Computação e Análise Numérica, foi a primeira linguagem de programação imperativa, criada para o IBM 704, entre 1954 e 1957, por uma equipe chefiada por John W. Backus.

Basic, sigla para *Beginners All-purpose Symbolic Instruction Code* (Código de Instrução Simbólico para Todos os Propósitos de Iniciantes): criada com fins didáticos, pelos professores John George Kemeny e Thomas Eugene Kurtz, em 1964, no Dartmouth College. Também é o nome genérico de uma extensa família de linguagens de programação derivadas do Basic original.

C: compilada, estruturada, imperativa, processual, de alto nível e padronizada. Foi criada em 1972, por Dennis Ritchie, no AT&T Bell Labs, como base para o desenvolvimento do sistema operacional UNIX (escrito em Assembly originalmente).

C++: de alto nível, com facilidades para o uso em baixo nível, multiparadigma e de uso geral. Desde os anos 1990, é uma das linguagens comerciais mais populares, mas disseminada também na academia por seu grande desempenho e base de utilizadores. Foi desenvolvida por Bjarne Stroustrup (primeiramente, com o nome C with Classes, que significa C com classes, em português), em 1983, no Bell Labs, como um adicional à linguagem C.

Roger S. Pressman é reconhecido internacionalmente como uma das maiores autoridades em engenharia de softwares. Trabalha como desenvolvedor, professor, escritor e consultor.

Roger S. **Pressman** indica sete áreas ou categorias potenciais de aplicação de softwares em seu livro Engenharia de Software (PRESSMAN, 2006). Para saber mais sobre essas sete áreas, consulte o quadro abaixo.

1.4. Processo de desenvolvimento

Na área de tecnologia da informação, desenvolvimento de sistemas significa o ato de elaborar e implementar um programa, ou seja, entender as necessidades dos usuários e transformá-las em um produto denominado software, de acordo com a definição de **Ana Regina Cavalcanti da Rocha**, no seu livro *Qualidade de Software: Teoria e Prática* (DA ROCHA, 2004).

Ana Regina Cavalcanti da Rocha é mestra e doutora em Informática; atua na área de Ciência da Computação, com ênfase em Engenharia de Software.

Os processos de desenvolvimento são essenciais para o bom andamento do projeto de um software, assim como a compreensão do sistema como um todo. Quanto mais aumenta a complexidade dos sistemas, mais difícil se torna a sua visibilidade e compreensão, portanto, sem um processo bem definido o projeto tem grande chance de insucesso.

O processo envolve atividades necessárias para definir, desenvolver, testar e manter um software. Exige inúmeras tentativas de lidar com a complexidade e de minimizar os problemas envolvidos no seu desenvolvimento. E devem ser levados em consideração fatores críticos: entrega do que o cliente deseja, com a qualidade, o prazo e o custo acordados (**PMI**, 2004).

A terceira edição do PMBOK® - Guia do Conjunto de Conhecimentos em Gerenciamento de Projetos saiu em 2004. É uma publicação do PMI (Project Management Institute, o Instituto de Gerenciamento de Projetos, em português), para identificar e descrever as boas práticas de projetos que agreguem valor e sejam fáceis de aplicar.

Para assegurar qualidade e padrão aos projetos, devem ser seguidos modelos de desenvolvimento de software: em cascata, iterativo ou incremental e prototipagem. Também é importante levar em conta o ciclo de vida de um programa, desde a sua concepção até a sua extinção. Logo, é necessário saber quanto tempo o software ficará em funcionamento e quais os benefícios gerados durante esse período, sempre tendo em mente o retorno do investimento (**FOWLER**, 2009).

Chad Fowler, autor de vários livros, é um dos mais competentes desenvolvedores de software do mundo. Trabalhou para várias das maiores corporações do planeta. Vive na Índia, onde mantém um centro internacional de desenvolvimento.

É crucial entender perfeitamente os entraves envolvidos no desenvolvimento de um software. Embora não exista uma conduta padrão, é sempre bom partir do princípio de que se está em busca de soluções para os problemas do cliente.

Com a definição de objetivos, atividades e participantes, consegue-se a excelência no desenvolvimento de software, pois há gestão, controle e padronização do processo. Tudo isso diminui os riscos de problemas com cronograma, treinamento, qualidade e aceitação do sistema pelos usuários. Vejamos quais são os objetivos, as atividades e os participantes.

1.4.1. Objetivos

Definição: alinhar todas as atividades a executar no decorrer de todo o projeto e, assim, desenvolver tudo o que foi previsto no seu escopo.

Planejamento: definir, em um cronograma, quando, como (quais os recursos de hardware e de software necessário) e quem irá realizar determinada tarefa.

Controle: ter sempre meios de saber se o cronograma está sendo cumprido e criar planos de contingência caso haja problemas no fluxo.

Padronização: seguir as mesmas metodologias por todos os envolvidos no projeto para não haver dificuldades de entendimento.

As sete áreas, por Pressman

- **Software básico ou de sistema:** conjunto de programas desenvolvidos para servir a outros sistemas, como os operacionais e os compiladores.
- **Sistemas de tempo real:** programas que monitoram, analisam e controlam eventos reais, no momento em que ocorrem. Recebem informações do mundo físico e, como resultado de seu processamento, enviam de volta informações de controle. Devem ter um tempo determinado de resposta para evitar efeitos desastrosos.
- **Sistemas de informação:** softwares da maior área de aplicação de sistemas, que engloba os programas de gerenciamento e acesso a bases de dados de informações de negócio.
- **Software de engenharia ou científico:** sistemas utilizados em áreas como astronomia, análise de resistência de estruturas, biologia molecular etc.
- **Sistemas embarcados ou software residente:** programas alojados nas ROM (Read-Only Memory, ou Memória Apenas de Leitura) e que controlam sistemas de baixo nível.
- **Software de quarta geração:** programas como processadores de texto, planilhas eletrônicas, jogos, aplicações financeiras e acesso a bases de dados.
- **Software de inteligência artificial (IA):** sistemas que utilizam algoritmos não numéricos para resolver problemas complexos, também conhecidos como sistemas baseados em conhecimento.

1.4.2. Atividades

Levantamento de requisitos: entender a necessidade do cliente e as regras do seu negócio (é a fase mais importante do desenvolvimento).

Análise de requisitos: definir o que fazer sob o ponto de vista de análise de sistemas.

Projeto: desenvolver o sistema já com cronograma, necessidades e riscos pre-estabelecidos.

Implementação: começar a usar um novo processo.

Testes: analisar se todas as funcionalidades solicitadas pelo cliente no levantamento de requisitos estão funcionando corretamente.

Implantação: disponibilizar os processos para utilização pelo usuário final.

1.4.3. Participantes

Gerentes de projeto: entram em contato com o cliente para levantar suas necessidades, assumindo a responsabilidade pelo cumprimento das fases de desenvolvimento e do cronograma.

Analistas de sistema: elaboram o projeto do sistema, utilizando UML (Unified Modeling Language, ou Linguagem de Modelagem Unificada), ferramentas de modelagem de processos, técnicas de análises de sistemas e técnicas de projetos de sistemas.

Ichak Adizes é o criador da metodologia que leva seu nome e visa ao diagnóstico e à terapêutica para mudanças organizacionais e culturais. O método está sendo aplicado – com muito sucesso – em organizações de 30 mil a 90 mil empregados de diversos países.

Arquitetos de software: definem a arquitetura em que o sistema funcionará (web, Windows, Linux etc.), com conhecimento dos pontos fortes e fracos de cada ambiente de acordo com as necessidades do cliente.

Programadores: codificam, em linguagem de programação, os requisitos do sistema, elaborados pelo analista de sistemas nos modelos UML.

Clientes: solicitam o desenvolvimento em entrevistas durante as quais serão definidos os requisitos do sistema.

Avaliadores de qualidade: realizam os testes do sistema, utilizando – ou não – ferramentas como o JTest.

1.5. Modelo de ciclo de vida

O ciclo de vida, em sistemas informatizados, tem as mesmas etapas do ciclo de vida de um ser humano (**ADIZES**, 1999) (figura 1). O namoro é considerado o primeiro estágio do desenvolvimento, quando o sistema ainda nem nasceu e existe apenas como uma ideia. Ainda não existe fisicamente, é apenas uma possibilidade. Trata-se, portanto, de um período em que se fala muito e se age pouco. O analista de sistemas tem, nessa etapa, uma função muito importante: entender o que o cliente deseja e, a partir daí, criar um compromisso com ele.

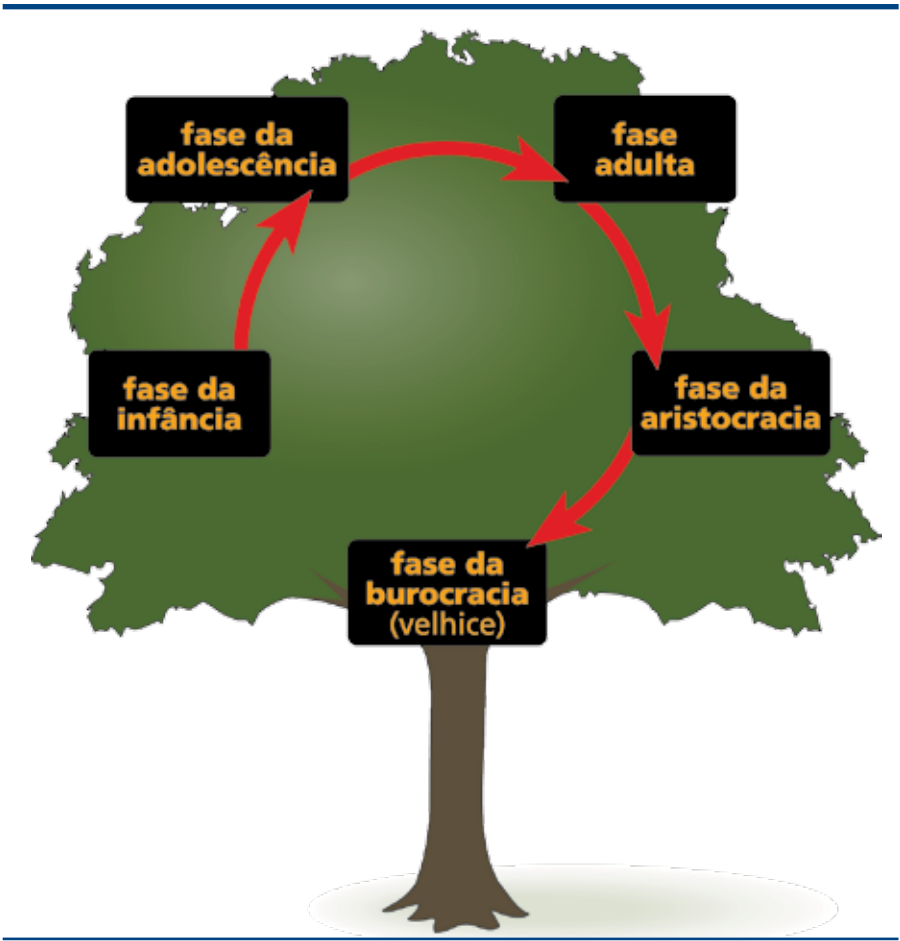



Figura 1
A imagem ao lado mostra que o ciclo de vida dos sistemas informatizados pode ser comparado ao dos seres humanos.

Já a fase da infância, também chamada de sistema-criança, conta com poucos controles formalizados e muitas falhas a serem transformadas em virtudes. Normalmente, o sistema é precário, faltam registros e informações, há resistência das pessoas em fazer reuniões de aprimoramento. Alguns chegam a acreditar que o sistema não vai funcionar.

A adolescência é o estágio do renascimento. Nessa etapa, ainda conforme Adizes, é eliminada grande parte dos erros encontrados na fase anterior. Essa transição é caracterizada por conflitos e inconsistências, muitas vezes causados pelos próprios usuários, os quais ainda não se comprometem a realizar as interações pertinentes. Mesmo percebendo a necessidade de delegar autoridade, mudar metas e liderança, os responsáveis enfrentam dificuldades, pois muitos usuários ainda acreditam que o antigo sistema era melhor.

A estabilidade, ou fase adulta, é o início do estágio de envelhecimento do sistema, ou seja, quando começa a se tornar obsoleto e surgem outros melhores. Um sintoma bem visível é a perda de flexibilidade. Todos começam a achar que ele não funciona tão bem, atribuindo-lhe falhas. É um estágio marcado pelo fim do crescimento e início do declínio.

A aristocracia – também batizada de meia-idade ou melhor idade – é a fase da vaidade, do vestir-se bem e de falar bem. Maria Aparecida Maluche explica que a ênfase está em como as coisas são feitas e não no porquê (MALUCHE, 2000).

É aí que se inicia a etapa de declínio total do sistema, na qual o nível de inovação é baixo e tudo deixa a desejar.

Na fase da burocracia, ou velhice, o sistema perde a funcionalidade e a elasticidade. Com isso, ninguém mais tem confiança nele. Muitos percebem a situação, mas ninguém faz nada, culpando o sistema por todos os erros e falhas na organização. O declínio se intensifica e, mesmo que permaneça em uso por alguns anos, a decadência prossegue, até a morte do sistema (ADIZES, 1999).

As agendas ficam superlotadas, começa-se a perder o controle de prazos e de qualidade. Ao mesmo tempo que se assume muitos compromissos, comete-se falhas. Por isso, deve-se ficar atento às reclamações dos clientes e tentar, de qualquer maneira, atender às necessidades percebidas.

A sequência de etapas

1. **Definição dos objetivos:** finalidade do projeto e sua inscrição dentro de uma estratégia global.
2. **Análise das necessidades e viabilidade:** identificação, estudo e formalização do que o cliente precisa e do conjunto de entraves.
3. **Concepção geral:** elaboração das especificações da arquitetura geral do software.
4. **Concepção detalhada:** definição precisa de cada subconjunto do software.
5. **Codificação, aplicação ou programação:** tradução em linguagem de programação das funcionalidades definidas nas fases de concepção.
6. **Testes unitários:** verificação individual de cada subconjunto do software, aplicados em conformidade com as especificações.
7. **Integração:** certificação e documentação da intercomunicação dos diferentes elementos (módulos) do software.
8. **Qualificação ou receita:** checagem da conformidade do software às especificações iniciais.
9. **Documentação:** registro das informações necessárias para a utilização do software e para desenvolvimentos anteriores.
10. **Produção:** colocação do sistema em operação para o cliente.
11. **Manutenção:** ações corretivas (manutenção corretiva) e evolutivas (manutenção evolutiva) no software.

O ciclo de vida de um software (*software lifecycle*) designa todas as etapas do seu desenvolvimento, da concepção à extinção. Essa segmentação tem por objetivo definir pontos intermediários que permitam checar a conformidade do sistema com as necessidades expressas no escopo do projeto e verificar o processo de desenvolvimento.

Segundo Ana Regina Cavalcanti da Rocha, em geral o ciclo de vida do software compreende, no mínimo, onze atividades (DA ROCHA, 2004). A sequência e a presença de cada uma delas no ciclo de vida dependem da escolha do modelo a ser adotado pelo cliente e pela equipe de desenvolvimento.

Os modelos de ciclo de vida descrevem as etapas do processo de desenvolvimento de software, assim como as atividades a serem realizadas em cada uma delas. A definição desses estágios permite estabelecer pontos de controle para a avaliação da qualidade e da gestão do projeto. Veja o quadro *A sequência das etapas*.

1.5.1. Modelo em cascata ou *waterfall*

Cascata ou *waterfall* (em inglês) é um dos mais simples e conhecidos modelos para o desenvolvimento de software. Criado em 1966 e formalizado em 1970, tem como principais características (FOWLER, 2009):

- Projetos reais raramente seguem um fluxo sequencial.
- Assume que é possível declarar detalhadamente todos os requisitos antes do início das demais fases do desenvolvimento (podendo ter a propagação de erros durante as fases do processo).
- Uma versão de produção do sistema não estará pronta até que o ciclo de desenvolvimento termine.

As fases são executadas sistematicamente, de maneira sequencial, conforme sugere a figura 2.

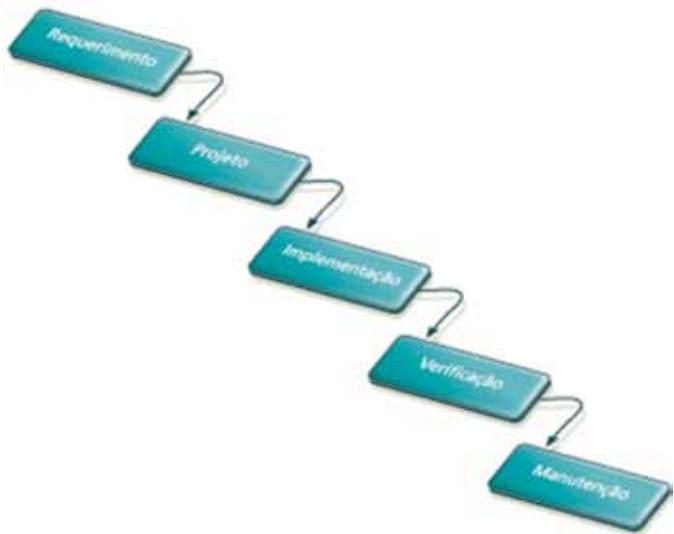



Figura 2
As fases sequenciais do Modelo cascata.

Requerimento (Análise e Definição de Requisitos): as funções, as restrições e os objetivos do sistema precisam ser definidos, via consulta aos usuários, para que sejam elaborados os detalhes específicos.

Projeto (Sistemas e Software): o processo deve agrupar os requisitos de hardware e software, assim como identificar e descrever as abstrações fundamentais do sistema e as suas relações.

Implementação (Testes de Unidade): o projeto é posto à prova como um conjunto de programas – ou de suas partes –, com o teste de cada item, para verificar se aquela unidade atende à sua especificação.

Verificação (Integração e Teste de Sistemas): antes da entrega do software ao cliente, as unidades ou programas individuais são integrados e testados como um sistema completo, de maneira a assegurar que todos os requisitos estejam contemplados.

Manutenção (Operação): o sistema é instalado e colocado em operação, sendo detectados e corrigidos erros não descobertos antes, o que melhora a implementação e também permite descobrir novos requisitos.

O sistema só é entregue ao usuário depois que os desenvolvedores observarem um resultado satisfatório na fase de verificação, com certo grau de confiança. Mesmo assim, o software ainda poderá ter alterações, principalmente para correção de falhas que só os usuários conseguem detectar no dia a dia e também para mais bem adaptá-lo ao ambiente ou ainda por problemas de desempenho. Se houver transformações no negócio da empresa, também será preciso realizar mudanças. A manutenção do software (última etapa proposta pelo modelo em cascata) pode requerer, portanto, voltar a atividades das fases anteriores do ciclo. É desse jeito que se garante a melhoria contínua do produto.

1.5.2. Modelo em cascata evolucionário ou *waterfall* evolucionário

As metodologias baseadas no modelo *waterfall* presumem que uma atividade deve ser concluída antes do início da próxima. Devido às limitações, houve reflexões no sentido de utilizar o modelo de maneira mais flexível. Assim, em uma situação extrema, aconteceriam simultaneamente todas as atividades do ciclo de vida em cascata, enquanto em outra circunstância também limite haveria o uso da abordagem sequencial, ou seja, concluir inteiramente uma fase antes de partir para a seguinte (FOWLER, 2009). A proposta é evoluir com base em protótipo inicial (figura 3). É fundamental que os requisitos sejam muito bem compreendidos. Mesmo com uma completa visão das especificações iniciais, é preciso trabalhar com o cliente durante todo o desenvolvimento. Todos os conceitos e ideias vão sendo materializados em requisitos, durante o processo, até que se chegue ao produto idealizado.

Conceitos e ideias vão sendo materializados, à medida que o trabalho evolui, até chegar ao produto desejado. O objetivo é trabalhar com o cliente durante toda a fase de desenvolvimento, com os requisitos muito bem compreendidos. Entre as vantagens dessa abordagem estão a possibilidade de antecipar o produto final para avaliação do cliente, de manter uma comunicação entre os desenvolvedores

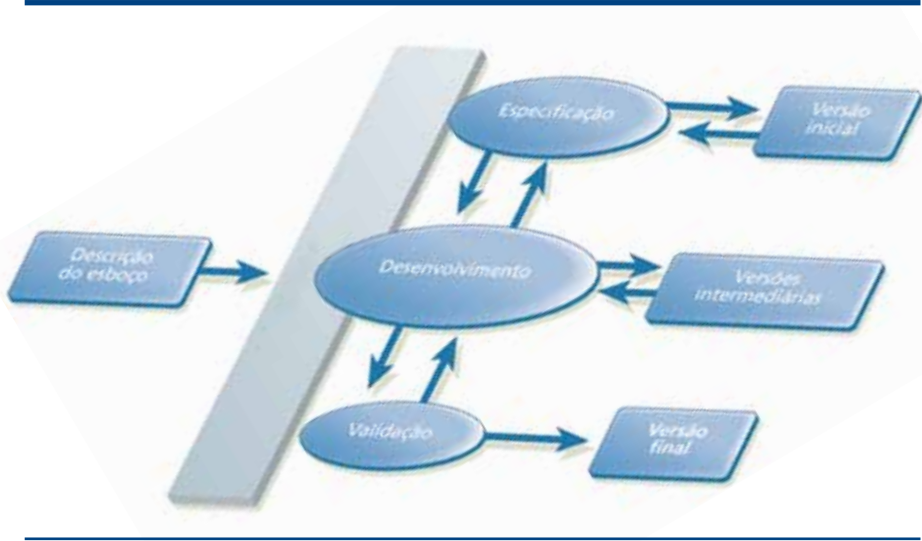



Figura 3
O modelo evolucionário.

e usuários para solucionar problemas técnicos e de começar precocemente o treinamento dos usuários. Isso antes de concluir as suas diferentes versões (inicial, intermediária e final), com espaço para readequações devidamente avalizadas pelo usuário. O modelo se baseia, então, no conceito da implementação inicial, cujo resultado é analisado – e comentado – pelo usuário. A partir desse *feedback*, o sistema é aprimorado por meio de muitas versões, até o seu desenvolvimento completo. A especificação, o desenvolvimento e a validação são executados concomitantemente para possibilitar um retorno rápido.

A abordagem mais radical do ciclo de vida em cascata é aquela em que todas as atividades se desenvolvem paralelamente, desde o início do projeto. Ao contrário, a mais conservadora é a que preconiza completar toda atividade N antes de iniciar a próxima N + 1. Mas há um número infinito de opções entre esses extremos. Segundo Edward **Yourdon**, a escolha entre as diversas possibilidades disponíveis depende de uma série de variáveis como é possível ver no quadro abaixo, que incluem do nível de estabilidade do usuário às exigências de produção (YOURDON, 1995).

Edward Yourdon é desenvolvedor de metodologias de engenharia de software, além de autor de 25 livros sobre computadores, incluindo best-sellers.

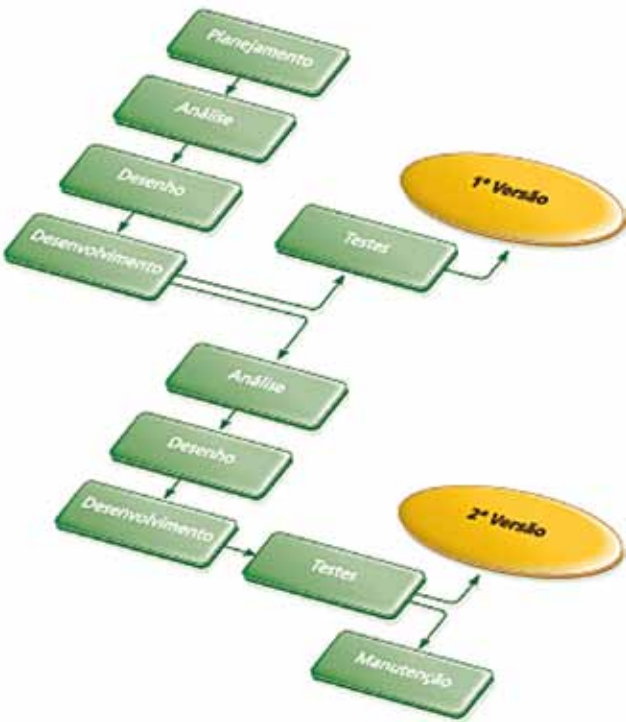
Variáveis de Yourdon

- **Nível de estabilidade do usuário:** se ele é instável ou inexperiente e não sabe definir suas reais necessidades, é preciso usar uma abordagem mais radical.
- **Nível de urgência na produção de resultados tangíveis e imediatos:** se o prazo é curto, por questões políticas ou outras pressões externas, justifica-se assumir uma abordagem radical. Nesse caso, é preferível ter 90% do sistema completo na data especificada do que 100% das etapas de análise e projeto.
- **Exigências para a produção de cronogramas, orçamentos, estimativas etc.:** a maior parte dos grandes projetos requer estimativas de pessoal, recursos de computação e outros detalhes, e tudo isso depende de levantamento e análise. Portanto, quanto mais detalhadas e precisas forem as estimativas, é provável que seja necessária uma abordagem conservadora.

1.5.3. Modelo incremental

O modelo incremental resulta da combinação do linear (em cascata) com o de protótipos (evolutivo). O seu desenvolvimento é dividido em etapas, denominadas incrementos, os quais conduzem aos aprimoramentos necessários, até que se chegue à versão final (FOWLER, 2009). Em cada incremento ocorre todo o ciclo de desenvolvimento de software, do planejamento aos testes. Por essa razão, ao final de uma etapa, já é possível obter um sistema totalmente funcional, embora não contemple todos os requisitos do produto final (figura 4).

Entre as vantagens dessa metodologia está justamente a possibilidade de definir melhor os requisitos, que muitas vezes não ficam muito claros no início do processo. O primeiro passo é trabalhar o núcleo do sistema, ou seja, são implementados os requisitos básicos, sem os detalhes. Em seguida o produto é avaliado pelo usuário, para a detecção de problemas iniciais, os quais já podem ser corrigidos, caso contrário poderiam assumir grandes proporções ao surgirem apenas na versão final. Outro benefício é o fato de o cliente esclarecer os requisitos e as prioridades para os próximos incrementos, além de poder contar com as facilidades da versão já produzida. Assim, com a construção de um sistema menor (sempre muito menos arriscada do que o desenvolvimento de um sistema grande), se um grave erro é cometido, ou se existe uma falha no levantamento de requisitos, somente o último incremento é descartado. Ganha-se em performance, pois são mínimas as chances de mudanças bruscas nos requisitos, e em tempo de desenvolvimento.

Figura 4
O modelo incremental.

1.5.4. Modelo iterativo

O método iterativo foi criado também com base nas vantagens e limitações dos modelos em cascata e evolutivo. Mantém a ideia principal do desenvolvimento incremental, com o acréscimo de novas capacidades funcionais, a cada etapa, para a obtenção do produto final. No entanto, segundo Fowler, modificações podem ser introduzidas a cada passo realizado. Mais informações sobre o modelo interativo no quadro abaixo.

Uma iteração de desenvolvimento abrange as atividades que conduzem até uma versão estável e executável do software. Por esse motivo, pode-se concluir que ela passa pelo levantamento de requisitos, desenvolvimento, implementação e testes. Ou seja, cada iteração é como um miniprojeto em cascata no qual os critérios de avaliação são estabelecidos quando a etapa é planejada, testada e validada pelo usuário.

Um dos principais ganhos com a adoção desse modelo é a realização de testes em cada nível de desenvolvimento (bem mais fácil do que testar o sistema apenas na sua versão final). Isso pode fornecer ao cliente informações importantes que servirão de subsídio para a melhor definição de futuros requisitos do software.

Passos do modelo iterativo

1. implementação inicial
- desenvolver um subconjunto da solução do problema global
- contemplar os principais aspectos facilmente identificáveis em relação ao problema a ser resolvido

2. lista de controle de projeto
- descrever todas as atividades para a obtenção do sistema final, com definição de tarefas a realizar a cada iteração
- gerenciar o desenvolvimento inteiro para medir, em determinado nível, o quão distante se está da última iteração

3. iterações do modelo
- retirar cada atividade da lista de controle de projeto com a realização de três etapas: projeto, implementação e análise
- esvaziar a lista completamente

Levantamento de requisitos
Desenvolvimento
Implementação
Testes
Implantação

Figura 5
O modelo espiral.

Barry W. Boehm é considerado um guru da Universidade do Sul da Califórnia, da qual é professor emérito, e tem lugar garantido entre os estudiosos que mais influenciaram o pensamento econômico a respeito de software nos últimos 40 anos. Ele se graduou em Harvard em 1957, é mestre e Ph.D. em Matemática e foi diretor do Software and Computer Technology Office do Departamento de Defesa dos Estados Unidos. Desde 1964 Boehm escreveu seis livros, entre eles *Software Risk Manager*, publicado pela IEEE Computer Society Press em 1989.

1.5.5. Modelo espiral

Criado por Barry **Boehm**, em 1988, o modelo espiral (figura 5) permite que as ideias e a inovação sejam verificadas e avaliadas constantemente. Isso porque, a cada interação, existe a volta da espiral que pode ser baseada em um modelo diferente e pode ter diferentes atividades. Esse padrão caracteriza-se, portanto, pelo desenvolvimento em sequência, aumentando a complexidade do processo conforme se chega mais próximo do produto final.

Em cada ciclo da espiral, uma série de atividades é realizada em uma determinada ordem, como comunicação com o cliente, planejamento, análise de riscos e avaliação dos resultados (FOWLER, 2009).

O modelo espiral é cíclico. Considerado um metamodelo, abrange diferentes padrões, desde o cascata até vários tipos de protótipos. Com isso, é possível prever os riscos e analisá-los em várias etapas do desenvolvimento de software.

O PMBOK define Gerência de Risco como todos os processos necessários para identificar, analisar, responder, monitorar e controlar os riscos de um projeto. E Dalton Valeriano, no seu livro Moderno Gerenciamento de Projetos, reforça o conceito ao afirmar que essa gestão deve ser constante durante todo o projeto, tendo como objetivos maximizar a probabilidade de riscos favoráveis e minimizar os negativos (VALERIANO, 2005).

1.6. Riscos

O risco do projeto é um evento ou uma condição incerta que, se ocorrer, terá um efeito positivo ou negativo sobre, pelo menos, um objetivo do projeto, como tempo, custo, escopo ou qualidade. E o modelo espiral foi a primeira proposta de inclusão de **Gerência de Risco** no ciclo de vida de desenvolvimento de software. A interatividade e os riscos, portanto, são as suas principais características (PMBOK, 2004). Aliás, como bem lembra Tom Gilb: "Se você não atacar os riscos [do projeto] ativamente, então estes irão ativamente atacar você".

1.6.1. Atividades do gerenciamento de riscos

Segundo o PMBOK, 2004, o processo de gerenciamento de riscos para desenvolvimento de software é composto por quatro atividades (figura 6):

1. Identificação dos riscos: reconhecimento do projeto, do produto e dos riscos de negócio envolvidos no desenvolvimento do software.

2. Análise dos riscos: estudo das possibilidades e das consequências da ocorrência dos riscos, com determinação de valores qualitativos para as possibilidades (muito baixo, baixo, moderado, alto e muito alto) e para as consequências (insignificante, tolerável, sério e catastrófico).

3. Planejamento do risco: medição de cada risco e desenvolvimento de uma metodologia de gerenciamento dentre algumas opções: refutar o risco quando a probabilidade é reduzida; minimizar o risco quando há baixo impacto no projeto de desenvolvimento de software ou no produto final; ou planos de contingência para eliminar o risco quando eles se tornarem realidade.

4. Monitoramento do risco: identificação e avaliação regular de cada risco para tomada de decisão quanto à diminuição, estabilidade ou aumento da possibilidade de ocorrência do evento; assim como discussão em cada reunião de avaliação do progresso do projeto sobre a permanência efetiva dos esforços para manter cada risco sob controle.

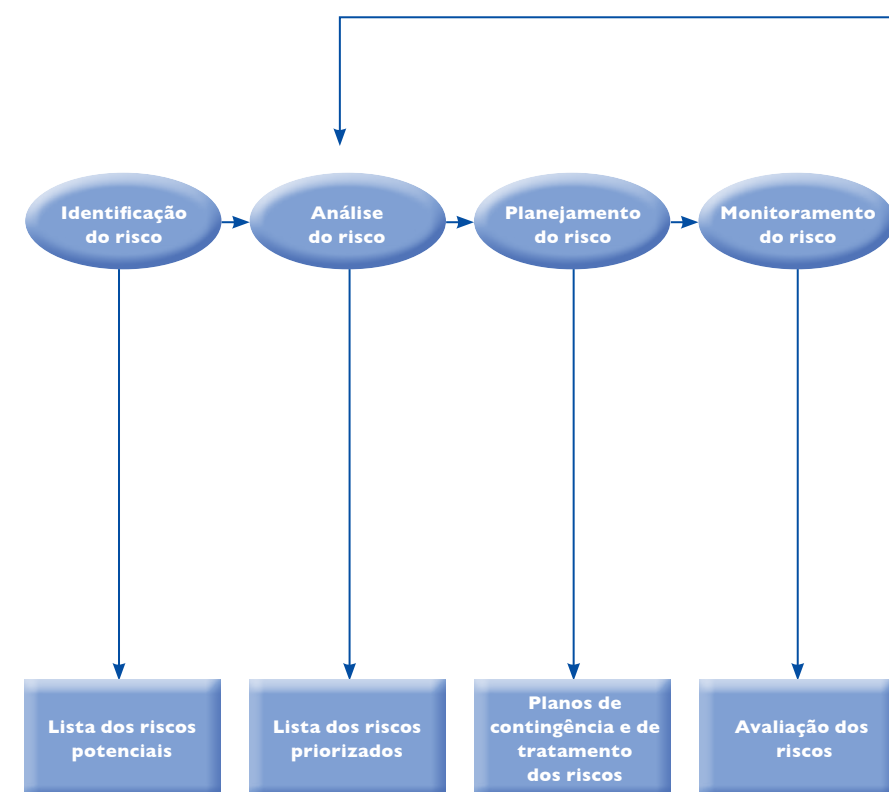



Figura 6
Atividades do processo de gerenciamento de riscos.

Para facilitar o desenvolvimento de um software, podem ser utilizadas ferramentas que auxiliem na construção de modelos, na integração do trabalho de cada membro da equipe, no gerenciamento do andamento do desenvolvimento etc. Os sistemas utilizados para dar esse suporte são normalmente chamados de CASE, sigla em inglês para Computer-Aided Software Engineering (Engenharia de Software de Suporte Computacional). Além dessas ferramentas, outros instrumentos importantes são aqueles que fornecem suporte ao gerenciamento, como desenvolvimento de cronogramas de tarefas, definição de alocações de verbas, monitoramento do progresso e dos gastos, geração de relatórios de gerenciamento etc.

1.7. Prototipagem

Fazer um protótipo de software garante a possibilidade de examinar antecipadamente os requisitos do programa (figura 7). Ou seja, **desenvolve-se** um exemplar simplificado, de forma rápida, para que as questões relativas a requisitos sejam entendidas ou esclarecidas. Com ele é possível melhorar muito a comunicação com o cliente, pois, primeiramente, ele é ouvido e, então, é feito um desenho e a construção do modelo. E é só depois de cumpridas essas três tarefas que se dá a validação, o que aumenta, em muito, as chances de sucesso do projeto (FOWLER, 2009).

Portanto, ao criar um protótipo, previnem-se possíveis deficiências no projeto de desenvolvimento de software. Em um número grande de casos, o usuário fica insatisfeito com o produto acabado. Isso ocorre por não ter havido troca de informação suficiente entre cliente e desenvolvedor. A comunicação no levantamento de requisitos é falha. Um protótipo deve ser utilizado como instrumento de análise, com a finalidade de superar as dificuldades de comunicação existentes entre o analista de sistemas e o usuário do sistema.

A prototipagem é uma técnica que pode ser empregada em pequenos ou grandes sistemas, em que os requisitos não estão claramente definidos. Seu uso é aconselhável em projetos para os quais o usuário não saiba exatamente o que deseja (FOWLER, 2009).

Alguns fatores devem ser levados em conta. Um protótipo é um esboço de alguma parte do sistema e a prototipagem é uma técnica complementar à análise de requisitos, com o objetivo de assegurar que as demandas foram bem entendidas. Já a construção desses protótipos utiliza ambientes com facilidades para a construção de interface gráfica, então alguns SGBDs (Sistemas Gerenciadores de Banco de Dados) também fornecem ferramentas para a concepção de telas de entrada e saída de dados.

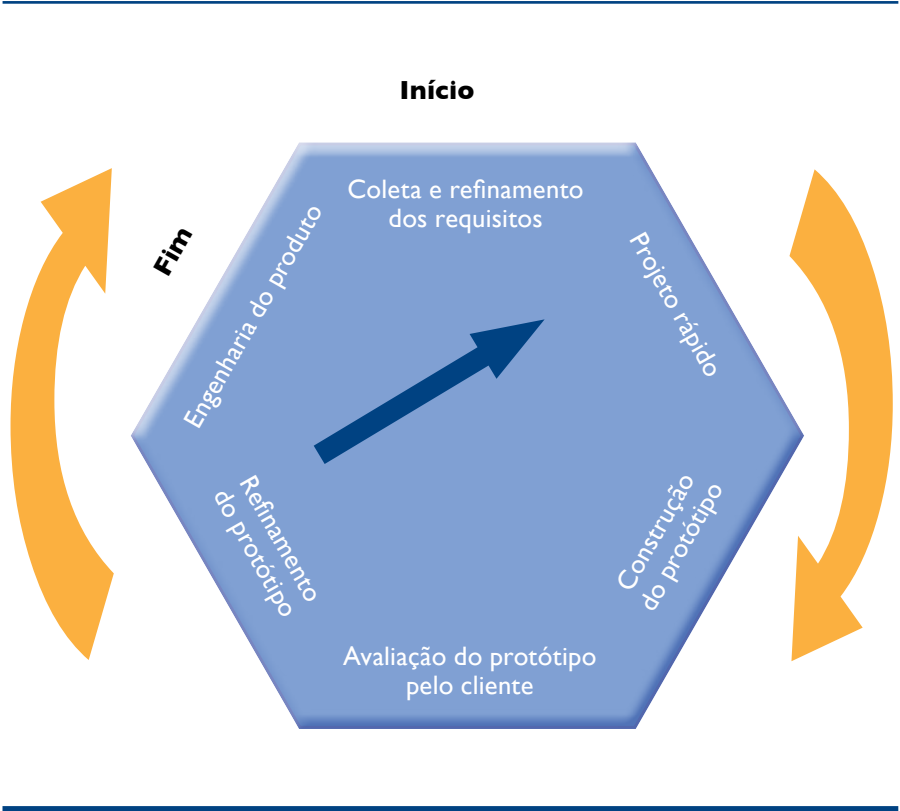



Figura 7
As fase da prototipagem.

Essa técnica é frequentemente aplicada quando existem dificuldades no entendimento dos requisitos do sistema e/ou quando existem requisitos que precisam ser mais bem entendidos.

Após o Levantamento de Requisitos (LR) – tema do próximo tópico deste livro –, um protótipo já pode ser construído e usado na validação. Os usuários fazem suas críticas e são feitas correções. Esse processo de revisão e refinamento continua até que o sistema seja aceito pelos usuários. A partir daí, é descartado ou utilizado apenas como uma versão inicial do sistema.

Importante: a prototipagem não substitui a construção de modelos do sistema. Trata-se de uma técnica complementar. E os erros detectados na sua validação devem ser utilizados para modificar e refinar o programa.

1.8. Levantamento ou especificação de requisitos

Obter qualidade nos processos e produtos de engenharia de software não é uma tarefa fácil, pois são vários os fatores que dificultam o alcance desse objetivo. No entanto, nada é mais decepcionante do que desenvolver um sistema que não satisfaça as necessidades dos clientes. Grandes volumes de recursos são gastos e, em muitos casos, os clientes sentem-se frustrados com a versão final do software encomendado. Muitos desses problemas são derivados da falta de atenção na hora de definir e acompanhar a evolução dos requisitos durante o processo de construção do sistema (DA ROCHA, 2004).

Por esse motivo, é necessário realizar o levantamento dos requisitos do sistema. As funcionalidades e os recursos devem ser definidos pelo usuário do sistema, isto é, pela pessoa da empresa que necessita de um programa para solucionar um determinado problema. Então, fica claro que o usuário está no centro de levantamento de requisitos do software. É quem conhece as necessidades a serem supridas pelo sistema e, por isso, precisa ser informado de sua importância no processo.

A falta de cuidado com o levantamento dos requisitos de um sistema pode gerar problemas muito sérios, como: resolução de um problema errado ou de forma errada; funcionamento diferente do esperado; complexidade na utilização e entendimento por parte dos usuários e alto custo (SWEBOK, 2004 – *Guide to the Software Engineering Body of Knowledge*). Tudo isso mostra a importância de empregar bem o tempo para entender o problema e seu contexto e obter, assim, os requisitos certos na primeira vez.

1.8.1. O que é um requisito

Requisito é a condição ou a capacidade que um sistema deve atender. Segundo uma das normas padrões do IEEE (1220, de 1994), um requisito é uma sentença identificando uma capacidade, uma característica física ou um fator de qualidade que limita um produto ou um processo. Isso significa uma condição

ou uma capacitação que um produto (no caso, um software) ou um serviço precisa atender ou ter para satisfazer um contrato, um padrão, uma especificação ou outro documento formalmente estabelecido entre as partes.

Os requisitos estão associados aos principais problemas do desenvolvimento de software, pois, quando não refletem as reais necessidades dos usuários, estão incompletos e/ou inconsistentes. Existem mudanças em requisitos já acordados, e a dificuldade para conseguir um entendimento entre usuários e executores é um dos principais problemas relatados, capaz de provocar retrabalho e, consequentemente, atrasos no cronograma, custos acima do orçamento e insatisfação do cliente (SWEBOK, 2004). Veja as categorias de requisitos no quadro abaixo.

Características

Os requisitos também são agrupados por suas características, em uma gama que inclui desde os necessários aos concisos e completos, entre outros.

Necessário: indica que, se retirado, haverá uma deficiência no sistema. Isto é, o programa não atenderá plenamente às expectativas do usuário. Não deve haver requisitos que seriam apenas interessantes, caso existissem. Ou o requisito é necessário ou é dispensável. Deve ser levado em consideração que cada requisito aumenta a complexidade e o custo do projeto, logo, não podem ser introduzidos de forma espúria.

Não ambíguo: tem apenas uma interpretação. É muito importante prestar atenção na linguagem utilizada para não causar duplo entendimento.

Verificável: não é vago nem genérico e deve ser quantificado para permitir a verificação de uma das seguintes formas: inspeção, análise, demonstração ou teste.

Conciso: para não gerar confusão, cada requisito define, de maneira clara e simples, apenas uma única exigência a ser desenvolvida. Para ser conciso, o requisito não inclui explicações, motivações, definições ou descrições do seu uso. Tais detalhes podem estar em outros documentos.

Alcançável: é realizável a um custo definido.

Completo: não precisa ser explicado ou aumentado, garantindo capacidade suficiente do sistema.

Consistente: não contradiz nem mesmo duplica outro requisito e utiliza os termos na mesma forma que a adotada nos outros requisitos.

Ordenável: tem uma ordem por estabilidade, importância ou ambos.

Aceito: acolhido pelos usuários e desenvolvedores.

DICAS PARA A ESCRITA

Algumas técnicas para escrever requisitos de software:

- **Preferir sentenças curtas.**
- **Utilizar verbos no futuro.**
- **Para cada requisito, avaliar se a definição determina se esse requisito está pronto ou não.**
- **Garantir que todos os requisitos podem ser verificáveis e o modo de fazer essa verificação.**
- **Verificar os requisitos agregados ou inter-relacionados.**
- **Estabelecer sempre o mesmo nível de detalhamento em todos os requisitos.**

Categorias de requisitos

- **Funcionais:** descrevem as funcionalidades e os serviços do sistema e dependem do tipo de software a ser desenvolvido, do número de usuários esperado e do padrão do sistema no qual o programa será utilizado. Exemplos: o sistema deve oferecer diversas maneiras de visualizar os dados, de acordo com o perfil do usuário; os relatórios devem estar disponíveis para impressão de acordo com o nível hierárquico de cada usuário etc.

- **Não funcionais:** definem as propriedades e as restrições do sistema, podendo especificar, por exemplo, quais linguagens de programação, bancos de dados e métodos de desenvolvimento serão utilizados. São mais críticos do que os requisitos funcionais, pois sua definição correta influencia diretamente a performance do sistema a ser desenvolvido.

- **De domínio:** originam-se do domínio da aplicação do sistema e refletem as suas características desse domínio. Podem ser requisitos funcionais ou não funcionais; requisitos funcionais novos; restrições sobre requisitos existentes ou sobre computações específicas.

Tipos

Seja qual for o sistema, há várias formas de atendê-lo. Dependendo das necessidades que se apresentam existem vários tipos de requisito:

- **Do usuário:** é a característica ou o comportamento que o usuário deseja para o software ou para o sistema como um todo. São descritos pelo próprio usuário ou por um analista de sistemas, a partir de um levantamento de dados com o cliente.

- **Do sistema:** comportamento ou característica que se exige do sistema como um todo, incluindo hardware e software. São normalmente levantados por engenheiros de software ou analistas de sistemas, que devem refinar os requisitos dos usuários, transformando-os em termos de engenharia de software.

- **Do software:** comportamento ou característica exigido do software. São normalmente levantados por analistas de sistemas com o objetivo de comparar todos os requisitos com aqueles que possuem real relevância.

Figura 8
Imprimir a Nota Fiscal.

EXEMPLOS DE REQUISITOS

1. O programa deve imprimir nota fiscal e fazer a integração com o sistema da Nota Fiscal Paulista e de vendas registradas (figura 8).

2. O software deve cadastrar novos produtos, gerar relatórios de vendas etc.

3. O sistema deve permitir a realização de vários orçamentos e análise, levando em consideração a condição de pagamento e o prazo de entrega.

Quando um requisito for funcional, deverá ser também:
Independente da implementação: define o que deve ser feito, mas não como deve ser feito.

1.8.2. Como devemos escrever requisitos

Normalmente as especificações de requisitos são escritas em linguagem natural (inglês ou português, por exemplo). Devem ser utilizadas técnicas de padronização para formato, estilo de linguagem e organização das informações. Tudo isso facilita a manipulação desse conjunto de requisitos.

1.8.3. Dependência de requisitos

Os requisitos não são independentes uns dos outros. E a maioria deles só pode ser implementada se outros forem implantados antes (eis o conceito de **requisitos predecessores**). Uma das atividades mais importantes da gerência de requisitos é manter esse relacionamento de dependência, que influenciará todo o processo de desenvolvimento do sistema. Para isso existem algumas soluções possíveis, como manter uma tabela de dependência de requisitos ou manter um banco de dados de requisitos que inclua relações de dependência. Existem alguns produtos no mercado especializados na gerência de requisitos que administram essas dependências.

Por exemplo, é impossível fazer um relatório de vendas sem registrá-las previamente no sistema: o requisito predecessor ao relatório de vendas é o registro das vendas.

É fundamental ter uma atuação constante e muito próxima aos usuários para que qualquer divergência em relação a requisitos ou a adaptações seja facilmente detectada e corrigida. Mas para tanto os usuários precisam estar comprometidos com o desenvolvimento do sistema e se sentirem donos dele. Se isso não ocorrer, há grandes probabilidades de o sistema não ser um sucesso.

1.8.4. Documentação de requisitos

A documentação de requisitos é uma atividade crucial para a engenharia de software, pois o documento gerado nessa fase é uma descrição oficial dos requisitos do sistema para cliente, usuários e desenvolvedores. Isso significa que todos os envolvidos no processo se basearão nesse documento para o desenvolvimento do sistema. Tal documento deve trazer muitas designações: especificação funcional, definição, especificação e responsáveis pelo requisito (FOWLER, 2009).

Ao documentar um requisito, deve-se ter em mente algumas questões que podem influenciar, em muito, todo o processo de desenvolvimento. É necessário se preocupar com a boa qualidade do documento, e alguns cuidados precisam ser observados no modo de escrever.

1. Quem escreve, para quem se escreve e qual o meio utilizado para escrever são três fatores diretamente relacionados à qualidade da documentação dos requisitos.

2. A adoção de linguagem natural – não técnica – no texto, complementada por diagramas, tabelas, fotos e imagens, facilita o entendimento daquilo que se deseja documentar.

3. O grau de detalhamento depende de quem escreve, da organização das informações, do propósito da documentação, entre outros quesitos.

4. A documentação serve para comunicar o que se pretende fazer em um determinado sistema e se ele se dirige a clientes, usuários, gestores e desenvolvedores do sistema.

5. A especificação tem de trazer os serviços que o sistema deve prestar, assim como suas propriedades e restrições impostas à operação e ao desenvolvimento.

6. O documento pode ser tanto sucinto e genérico quanto extenso e detalhado.

1.8.5. Métodos de identificação e coleta

Existem duas técnicas normalmente utilizadas para a identificação e coleta de requisitos: entrevista e reuniões de **JAD**, aquelas das quais participam usuários e analistas, para trabalhar na arquitetura de uma determinada aplicação. Ambas são consideradas as melhores práticas.

JAD (Joint Application Development) é um sistema de desenvolvimento de aplicações em grupo criado pela IBM. O objetivo é reduzir custos e aumentar a produtividade nesses processos.

1.8.5.1. Entrevista

Entre as técnicas mais importantes de identificação de requisitos está a **entrevista**, também presente em outras metodologias, pois, quase sempre, a coleta de informações exige a comunicação direta com os usuários e interessados. Tem como finalidade captar o pensamento do cliente para que se entenda o que é necessário desenvolver. Por essa razão, o entrevistador tem grande responsabilidade. Além de fazer as entrevistas, cabe a ele integrar diferentes interpretações,

Existem formas distintas de entrevista: por questionário, aberta e estruturada.

objetivos, metas, estilos de comunicação e terminologias adequadas aos diversos níveis culturais dos entrevistados.

Por questionário

O questionário é muito usado como técnica de entrevista, principalmente em pesquisas de mercado e de opinião. Sua preparação exige bastante cuidado, e alguns aspectos particulares do processo merecem destaque: emprego de vocabulário adequado para o público entrevistado; inclusão de todos os conteúdos relevantes e de todas as possibilidades de resposta; atenção aos itens redundantes ou ambíguos, contendo mais de uma ideia ou não relacionados com o propósito da pesquisa; redação clara e execução de testes de validade e de confiabilidade da pesquisa.

Uma das principais dificuldades que envolvem esse trabalho é o fato de as palavras possuírem significados distintos para pessoas diversas em contextos diferentes (culturais e educacionais, por exemplo). Em conversações corriqueiras, tais dificuldades de interpretação são esclarecidas naturalmente, mas, em entrevistas com questionários, o treinamento e o método utilizados proíbem essa interação. Por outro lado, tudo isso garante menos ambiguidade, uma das principais vantagens da entrevista por questionário.

Entrevista aberta

A entrevista aberta supre muitas das lacunas dos questionários, porém gera outros problemas. O entrevistador formula uma pergunta e permite ao entrevistado responder como quiser. Mesmo que o primeiro tenha a liberdade de pedir mais detalhes, não há como determinar exatamente o escopo da entrevista. Por isso essa técnica pode gerar dúvidas: as perguntas podem ser de fato respondidas ou a resposta faz parte do repertório normal do discurso do entrevistado? Existem muitas coisas que as pessoas sabem fazer, mas têm dificuldade para descrever, assim como há o conhecimento tácito, de difícil elucidação.

Os benefícios das entrevistas abertas são a ausência de restrições, a possibilidade de trabalhar uma visão ampla e geral de áreas específicas e a expressão livre do entrevistado. Há desvantagens também. A tarefa é difícil e desgastante, mas entrevistador e entrevistado precisam reconhecer a necessidade de mútua colaboração para que o resultado seja eficaz.

Também deve-se levar em conta a falta de procedimentos padronizados para estruturar as informações recebidas durante as entrevistas, uma vez que a análise dos dados obtidos não é trivial. É difícil ouvir e registrar simultaneamente: alguns fatos só tomam determinada importância depois de outros serem conhecidos e, aí, o primeiro pode não ter sido registrado. Por isso vale gravar toda a conversa e transcrevê-la, o que facilita a tarefa de selecionar e registrar o que é relevante para validar com o entrevistado posteriormente.

O relacionamento entre os participantes de uma entrevista deve ser baseado no respeito mútuo e em algumas outras premissas, tais como deferência ao conhecimento e habilidade do especialista; percepção de expressões não verbais; sensibilidade às diferenças culturais; cordialidade e cooperação.

TAXI/GETTY IMAGES

Figura 9
O sucesso da entrevista depende da escolha da pessoa certa para oferecer as respostas precisas.

Entrevista estruturada

A entrevista estruturada (figura 9) tem por finalidade coletar e organizar as informações sobre questões específicas, necessárias para o projeto. É fundamental realizar a entrevista com a pessoa certa, ou seja, quem realmente entende e conhece o negócio e os seus processos. E preparar muito bem o roteiro do que se quer obter, aprofundando o assunto.

A principal vantagem dessa técnica é a obtenção de respostas diretas com informações detalhadas. Todavia, como desvantagem, muitas vezes aspectos relevantes ao projeto de desenvolvimento de software precisam ser identificados antes da realização da entrevista. De maneira geral, requer muito mais trabalho prévio e, quanto melhor for feito, maiores serão as chances de realizar esse trabalho com eficácia.

Preparação

Para tudo que se faz na vida é preciso se preparar. Em relação às entrevistas não é diferente. Deve-se elaborar um roteiro coerente ao alcance dos objetivos do projeto e informar ao entrevistado tanto o propósito da conversa quanto sua importância no processo. Selecionar o entrevistado é outro ponto de atenção, pois somente ao final das entrevistas será possível ter uma visão clara e completa do problema a ser solucionado e das diversas formas de analisar e resolver.

A linguagem precisa se adequar ao entrevistado, levando em consideração seu perfil cultural e tomando muito cuidado com a utilização de termos técnicos,

muitas vezes corriqueiros para o entrevistador, mas eventualmente desconhecidos pelo respondente. Às vezes, por vergonha, essa pessoa não admite que não sabe do que se trata e responde de maneira equivocada. Daí a importância da coerência das perguntas e de observar, com atenção, o real entendimento dos entrevistados. Mais um ponto a cuidar: a programação e o cumprimento dos horários estabelecidos.

Realização

O objetivo central de uma entrevista é conseguir informações relevantes do entrevistado. Para isso é fundamental fazer com que ele não se limite a falar, mas que também reflita bastante sobre cada uma das suas respostas. Por isso o entrevistador deve deixar o respondente totalmente à vontade, sem pressa para acabar o trabalho. Veja o quadro *Questionário bem feito*.

É importante evitar interrupções ocasionadas por fatores externos (telefone, entra-e-sai de pessoas da sala etc.). Tudo isso pode levar o entrevistado a perder o foco.

Comportamento do entrevistador

O entrevistador deve demonstrar claramente as suas intenções para o entrevistado. Primeiramente, precisa estar vestido conforme os costumes da empresa para a qual está trabalhando, de maneira a não causar desconforto aos entrevistados. Até há pouco tempo, consultores e desenvolvedores de software usavam terno, normalmente escuro, intimidando entrevistados não habituados a esse tipo de vestuário.

Requisitos para a documentação

- Visão geral do sistema e dos benefícios decorrentes do seu desenvolvimento.
- Glossário explicativo dos termos técnicos utilizados.
- Definição dos serviços ou dos requisitos funcionais do sistema.
- Definição das propriedades do sistema (requisitos não funcionais), como viabilidade, segurança etc.
- Restrições à operação do sistema e ao processo de desenvolvimento.
- Definição do ambiente em que o sistema operará e as mudanças previstas nesse ambiente.
- Especificações detalhadas, incluindo modelos e outras ferramentas.

Certos movimentos também podem causar desconforto no interlocutor. Balançar os pés ou bater a caneta na mesa são gestos involuntários que podem tirar a atenção do entrevistado e do entrevistador. E, no caso de entrevistas longas, é necessário fixar um breve intervalo.

Documentação

Toda entrevista deve ser documentada logo após sua realização. Desse modo, têm-se os dados arquivados e facilmente recuperáveis para eventuais necessidades. Esse material deve conter informações mínimas, podendo ser ampliado, dependendo da necessidade de cada projeto de desenvolvimento:

- data, hora e local da entrevista;
- nome do entrevistador;
- cargo do entrevistador;
- nome do entrevistado;
- função do entrevistado e descrição do cargo;
- organograma do entrevistado (superior imediato, colegas do mesmo nível, subordinados);
- objetivo da entrevista;
- nomes e títulos de todos os outros presentes na entrevista;
- descrição completa dos fatos tratados e opiniões do entrevistado;
- transcrição da entrevista (opcional);
- conclusões tiradas com base nos fatos e nas opiniões apresentados;
- problemas do negócio levantados durante a entrevista;
- exemplos de relatórios, diagramas, documentos etc.;

DICA
É importante que o desenvolvedor tenha conhecimentos do negócio do cliente e resista a priorizar a tecnologia em relação às suas necessidades.

Questionário bem-feito

- Evitar palavras ambíguas ou vagas que tenham significados diferentes para pessoas distintas.
- Redigir itens específicos, claros e concisos e descartar palavras supérfluas.
- Incluir apenas uma ideia por item.
- Evitar itens com categorias de respostas desbalanceadas.
- Evitar itens com dupla negação.
- Evitar palavras especializadas, jargões, abreviaturas e anacronismos.
- Redigir itens relevantes para a sua pesquisa.
- Evitar itens demográficos que identifiquem os entrevistados.

• desenhos e diagramas elaborados a partir da entrevista (ou durante a conversa);
• comentários relevantes feitos pelo entrevistado;
• todos os números importantes: quantidades, volume de dados etc.

Perguntas para conclusão

Para finalizar a conversa é preciso obter a avaliação do entrevistado sobre a atividade realizada. Essa tarefa exige que ele responda a um formulário contendo as seguintes perguntas:

• A entrevista cobriu todo o escopo necessário?
• Foram feitas perguntas adequadas?
• O entrevistado era realmente a pessoa mais indicada para dar as respostas solicitadas?

1.8.5.2. Metodologia JAD (Joint Application Design)

Outra técnica importante de identificação e coleta de requisitos é o JAD (iniciais de Joint Application Design ou, literalmente, desenho de aplicação associada).

Caso o levantamento de requisitos não seja feito de maneira eficiente – sem a identificação das verdadeiras necessidades do usuário –, ao entregar o software o grupo de informática corre o risco de receber um *feedback* negativo do cliente. Isso porque a percepção desse cliente será de que não recebeu aquilo que solicitou.

Tal problema de comunicação pode ter diversas causas. A adoção de linguagem técnica por ambas as partes é uma das principais razões de erros no levantamento de requisitos, pois tanto o usuário quanto o analista de sistemas podem utilizar termos pertinentes exclusivamente à sua área. Por exemplo, profissionais do departamento financeiro possuem um conjunto de vocábulos técnicos (jargões) desconhecidos dos analistas de sistemas ou que podem levar a um entendimento dúbio. O desconhecimento dos desenvolvedores sobre a área de atuação é outro motivo de equívocos.

Figura 10
A metodologia JAD substitui as entrevistas individuais por reuniões entre usuários e desenvolvedores.

Recomenda-se, portanto, que esses profissionais tenham conhecimentos sobre o negócio do cliente, para estarem mais integrados ao sistema a ser desenvolvido. Também não se deve priorizar a tecnologia em detrimento das necessidades dos usuários e, consequentemente, da solução do problema do cliente. Na fase inicial do projeto, a dificuldade de comunicação entre clientes e equipe de desenvolvimento é a principal causa de defeitos encontrados no produto final (AUGUST, 1993).

Na tentativa de resolver os problemas de comunicação entre desenvolvedores e clientes (usuários), foram criados, no final da década de 1970, diversos métodos baseados na dinâmica de grupo.

O processo da entrevista

O processo de entrevista não se reduz ao ato em si. Por isso, é necessário pensar em todas as atividades que antecedem e sucedem a tarefa:

1. Determinação da necessidade da entrevista
2. Especificação do objetivo da entrevista
3. Seleção do entrevistado
4. Marcação da entrevista
5. Preparação das questões ou do roteiro
6. Entrevista propriamente dita
7. Documentação da entrevista, incluindo as informações obtidas
8. Validação com o entrevistado da transcrição da entrevista
9. Correção da transcrição
10. Aceitação por parte do entrevistado
11. Arquivamento

Roteiro para realização de entrevista

Para a realização de uma entrevista bem estruturada é preciso seguir um roteiro ou um script. Pode-se utilizar um modelo (template), retirando-se ou acrescentando-se itens sempre que necessário. E uma boa opção é estabelecer uma conversa informal antes da entrevista, para deixar o entrevistado mais à vontade. Depois é só seguir o roteiro:

1. Apresentar-se ao entrevistado, identificando-se e informando de qual projeto participa e qual a sua função (gerente de projetos, analista de sistemas etc.).
2. Informar ao entrevistado qual o motivo da entrevista e as razões pelas quais ele foi selecionado: é a melhor pessoa para auxiliar no levantamento de requisitos, por conhecer os procedimentos.
3. Deixar claro que o conhecimento e as opiniões do entrevistado são de extrema importância e serão muito úteis no processo de análise de requisitos.
4. Dizer ao entrevistado o que será feito com as informações levantadas.
5. Informar o entrevistado de que lhe será fornecida uma transcrição da entrevista, para que tenha oportunidade de ler e corrigir algum detalhe eventualmente equivocado e lhe assegurar que nenhuma informação será fornecida a outras pessoas até essa validação.
6. Determinar assuntos confidenciais ou restritos a serem tratados na entrevista.
7. Deixar claro que não haverá consequências negativas em função do resultado da entrevista.
8. Solicitar sempre a permissão do entrevistado para gravar a entrevista.
9. Se o entrevistado autorizar, deve-se iniciar a gravação com um texto de apresentação, para identificar a entrevista, informando o assunto e a data.
10. Avisar ao entrevistado quando o tempo estiver se esgotando e perguntar se ele gostaria de adicionar alguma informação.
11. No final, solicitar ao entrevistado que responda às perguntas de conclusão.
12. Concluir a entrevista de forma positiva, relatando brevemente o bom resultado alcançado.
13. Se necessário, marcar outra entrevista.
14. Despedir-se educadamente, agradecendo a atenção e o tempo dispensado.

Pela forma tradicional de desenvolvimento de software, os analistas de sistemas entrevistam usuários um após outro, área a área. Anotam tudo o que é dito e, somente em um segundo momento, as informações são consolidadas e validadas com os entrevistados. Após essa verificação, os dados são compilados em um documento de análise. Já ao se utilizar o JAD, as entrevistas individuais são substituídas por reuniões em grupo nas quais os envolvidos com o processo (usuários) participam ativamente ao lado da equipe de desenvolvimento.

Quando o método JAD é aplicado de forma correta, os resultados são excelentes. Além de atingir o objetivo de obter informações dos usuários, cria-se um ambiente de integração da equipe onde todos se sentem envolvidos e responsáveis por encontrar soluções. Esse comprometimento dos usuários faz com que eles se sintam "proprietários" do sistema e "colaboradores" no seu desenvolvimento, gerando, assim, maior aceitação do software quando este for implementado (AUGUST, 1993).

Capítulo 2

Modelo de entidade e relacionamento

- Conceitos
- Normalização
- Fases de um projeto utilizando ER

A necessidade de organizar informações acompanha a humanidade desde o início dos tempos. O pastor representava ovelhas por meio de pedras, enquanto os escribas ordenavam os textos nas estantes. Acontece que a população mundial cresceu, assim como a quantidade de informações disponíveis, e, desse modo, pesquisá-las tornou-se mais complexo, o que exigiu novas formas de organização.

Tente achar um livro na biblioteca (figura11) de sua escola sem saber como estão organizadas as estantes. Vá procurando estante por estante, livro por livro... Quanto tempo irá demorar? Agora, pense em fazer a mesma pesquisa na Biblioteca Nacional (do Rio de Janeiro) ou na Biblioteca do Smithsonian Museum, dos Estados Unidos, que são muito maiores que a de sua escola. Sem uma organização prévia fica muito demorado e trabalhoso obter as informações de que precisamos em nosso dia a dia.

E como a informática pode ajudar na organização? Muito e em todo esse processo, que inclui armazenamento, manutenção e consulta de informações, proporcionando agilidade, uniformidade e segurança em todas as suas fases. Ao juntarmos o conhecimento preexistente à velocidade de processamento e à capacidade de armazenamento de informações que a informática oferece, chegamos a modelos extremamente interessantes no que diz respeito à facilidade de uso, velocidade de acesso e de respostas, além de baixo custo de manutenção.

Depois de vários estudos, chegou-se a uma metodologia para projetar e construir bancos de dados otimizados, capazes de permitir o acesso mais rápido e consistente às informações, utilizando espaço cada vez menor de armazenamento. É sobre essa metodologia que falaremos neste capítulo, o modelo de entidade e relacionamento, que propõe definições e regras para projeto e implementação de bancos de dados, assim como a relação desses dados com as funcionalidades do sistema.

As bases do modelo de entidade e relacionamento começaram a ser lançadas quando Edgar Frank Codd definiu as principais implementações necessárias para o correto funcionamento de um banco de dados relacional usando, para isso, a teoria dos conjuntos, a álgebra e o cálculo relacional. O modelo ganhou mais corpo quando Donald D. Chamberlin e Raymond F. Boyce desenvolveram uma linguagem de consulta que facilitava o acesso e a manutenção de bancos de dados relacionais, oferecendo os recursos necessários para sua utilização em larga escala, o que atendia às necessidades do mercado. A contribuição de Peter Chen foi na definição de uma metodologia para modelagem de projetos de banco de dados, utilizando banco de dados relacionais (veja quadro *Os nomes por trás da tecnologia*).

A linguagem criada por Clamberlin e Boyce ganhou o nome de SQL, e somente em 1986 foi distribuída e aceita por praticamente todos os bancos de dados, tor-

Figura 11
Se soubermos como estão organizadas as estantes, podemos encontrar um livro facilmente, seja qual for o tamanho de uma biblioteca.

Figura 12
Não é possível encontrar um só livro em ambiente desorganizado.

ROBERT W. GINN/ALAMY/OTHER IMAGES

nando-se referência, com o lançamento do SQL padrão ANSI. A padronização foi necessária porque cada banco de dados, por questão de projeto ou facilidade de implementação, modificava os comandos da linguagem, a tal ponto que, hoje, se não fosse a padronização, provavelmente teríamos de reaprendê-la a cada mudança de sistema gerenciador de banco de dados. A linguagem SQL será um dos focos do terceiro capítulo deste livro.

Infelizmente, a padronização ainda não gerou uma linguagem com funções totalmente iguais, o que nos obriga, ao trocarmos de sistema gerenciador de banco de dados, a pesquisar, por exemplo, a função que retorna à data atual no SQL. Mas, com certeza, as diferenças entre elas são atualmente bem menores. Hoje em dia, o padrão ANSI está na versão SQL:2003.

Os estudos não pararam por aí. O modelo de entidade e relacionamento foi inegavelmente um marco na história da informática, utilizado em larga escala, mas avançou-se bastante depois dele. Hoje, por exemplo, existe a UML (Linguagem de Modelagem Unificada), outra técnica de modelagem, baseada na teoria de Orientação a Objetos (analisada no capítulo 4).

2.1. Conceitos

Para podermos utilizar as técnicas do modelo de entidade e relacionamento, necessitamos predefinir alguns de seus conceitos, de modo a facilitar seu entendimento.

Banco de dados

É um conjunto de informações inter-relacionadas sobre determinado assunto e armazenadas de forma a permitir acesso organizado por parte do usuário.

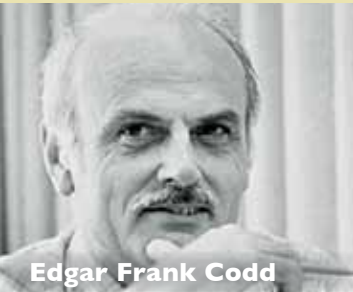
Bancos de dados relacional

São conjuntos de dados, relacionados entre si, que implementam as características do modelo de entidade e relacionamento.

Sistema gerenciador de bancos de dados (SGBD)

É um conjunto de programas que permite a implementação de bancos de dados,

Os nomes por trás da tecnologia

Edgar Frank Codd, Donald D. Chamberlin e Peter Pin Shan Chen são os pesquisadores que mais contribuíram para a criação do modelo de entidade e relacionamento. Em junho de 1970, Codd, matemático inglês, que na época trabalhava no laboratório da IBM em San José, na Califórnia, Estados Unidos, publicou um artigo decisivo na revista ACM – Association for Computer Machinery (Associação para Maquinária da Computação), intitulado *Relational Model of Data for Large Shared Data Banks* (*Modelo de dados relacional para grandes bancos de dados compartilhados*). Quatro anos depois, em maio de 1974, Chamberlin e Raymond F. Boyce, ambos engenheiros e cientistas da IBM, apresentaram o trabalho *SEQUEL – A Structured English Query Language* (*Linguagem de consulta estruturada em inglês*). Em março de 1976, Peter Chen, engenheiro elétrico e Ph.D. em Ciência da Computação, publicou, também na revista ACM, o artigo *The Entity-Relationship Model-Toward a Unified View of Data* (*O modelo de entidade e relacionamento, uma visão unificada dos dados*).

Edgar Frank Codd

Donald D. Chamberlin

Dr. Peter Chen

FOTOS: DIVULGAÇÃO

assim como o controle de acesso, o backup, a recuperação de falhas, a manutenção da integridade, a administração e a segurança dos dados que contém.

Modelo

Para a construção de modelos, é preciso abstrair, isto é, não se preocupar com todos os detalhes, mas apenas com os que se quer analisar e/ou sobre os quais se tem alguma dúvida.

Podemos definir um modelo como sendo um protótipo, em escala menor, do produto que queremos implementar ou da solução que queremos obter.
O **modelo** nos permite, com um custo muito menor (de tempo, dinheiro e trabalho), em comparação ao do desenvolvimento do produto final, analisar e desenvolver alguns aspectos que farão a diferença no produto final. Ou seja, no modelo podemos criar, testar funcionalidades novas e avaliar o projeto, com um baixo custo antes de sua implementação.

Abstração

ATENÇÃO
O Modelo ER e os Sistemas Gerenciadores de Bancos de Dados foram criados primeiramente para aceitar nomes em inglês, língua que não possui acentos em suas palavras. Apesar de hoje em dia a maioria dos SGBD´s aceitarem palavras acentuadas e até o uso do caracter espaço entre as palavras que nomeiam algum de seus componentes, por convensão, não utilizaremos espaços nem acentos para nominar os componentes de nossos modelos afim de não gerarmos problemas de implementação em qualquer que seja o SGBD, padronizarermos a implementação, evitando dúvidas do tipo Funcionário com ou sem acento, sem falar nas mudanças da língua.

Para exemplificar o que é abstração, vale acompanhar um exemplo bem corriqueiro e que muita gente vivencia. Quando olhamos para uma casa, podemos pensar em seu tamanho, sua localização, no número de quartos que a integram, na cor das paredes. Já um engenheiro civil, ao olhar para a mesma casa, pensará em como ela foi construída, se o material utilizado é de boa qualidade, se sua estrutura foi concebida para suportar seu tamanho. O pedreiro pensará na quantidade de tijolos, cimento, pedras, areia e ferro que foram necessários para construí-la. O jardineiro avaliará sua localização, o clima e sua posição em relação ao sol, além das melhores plantas para o jardim. O encanador pode refletir sobre qual é o tamanho da caixa d'água para garantir o abastecimento na casa e em quantos metros de encanamento de cada largura foram necessários para seu sistema hidráulico. O eletricista talvez pense sobre a metragem e a bitola dos fios empregados no sistema elétrico. O corretor de imóveis se concentra na metragem da casa, no número de cômodos, de vagas na garagem, na localização, no preço de aluguel ou venda (figura 13).

Cada um dos personagens do exemplo se fixou em detalhes diferentes sobre um mesmo projeto, a casa. Olhou para ela pensando em suprir as próprias necessidades, enfatizando as características mais importantes para atendê-las, e desprezou os demais detalhes, os quais, embora também façam parte da construção, não têm relevância particular.

É o que se chama de abstração: esses vários pontos de vista, essas diferentes possiblidades de análises sobre um mesmo objeto é o que se chama de abstração, uma característica fundamental para a construção de um bom modelo de entidade e relacionamento.

Modelo de entidade e relacionamento

Propõe definições e regras para o projeto e a implementação de bancos de dados, assim como a relação desses dados com as funcionalidades que esse sistema deve implementar. Sugere que, nas diversas fases de desenvolvimento do projeto, os modelos sejam refinados até que se chegue ao modelo final que, em modelagem ER, chamamos de projeto físico. Acrescentando-se a seguir o projeto físico do banco de dados, ele se juntará com as funciona-

Figura 13
Vários olhares sobre um mesmo tema.

lidades dos programas da aplicação, para só então chegar-se a uma solução completa de software. Há vários componentes do modelo ER. Vale, portanto, conhecer os principais, entre os quais se alinham: Entidade, Relacionamento, Atributo e Chaves.

Entidades

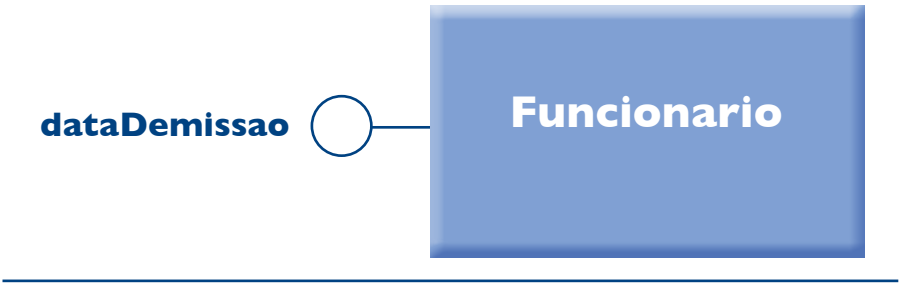
Entidades são abstrações do mundo real que contêm um conjunto de informações inter-relacionadas e coerentes. Estas informações são chamadas de atributos. Toda entidade possui um nome que a identifica, geralmente formado por um substantivo no singular. A representação gráfica de uma entidade é feita por um retângulo com seu nome no centro, como mostra a figura 14.

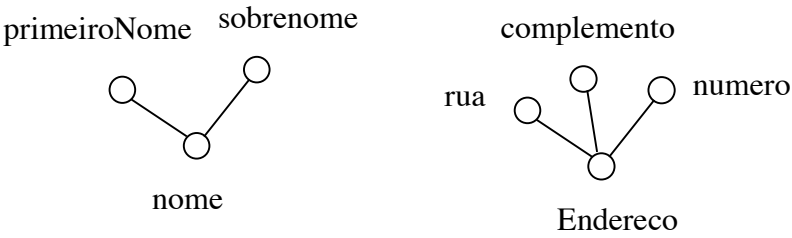
Figura 14
Entidade Funcionario.

Atributo

Atributo é cada informação que compõe uma entidade. Possui um nome, um tipo e um tamanho (número de caracteres). De modo genérico o tipo, pode ser nominado como texto, número, data, hora, etc. até que se saiba em qual sistema gerenciador de banco de dados este será implementado e então se atribua o tipo correto, pois cada SGBD possui suas particularidades em relação aos tipos de dados aceitos. Por exemplo os tipos real ou double.

O atributo pode ser representado no diagrama ER como um círculo, com o nome ao lado ou como uma elipse com seu nome, o qual é representado geralmente por um substantivo. Para evitar problemas de compatibilidade, deve começar com uma letra e não conter espaços e acentuação, mas pode incluir caracteres especiais como underline, entre outros (figura 15).

Figura 15
Atributo.

Há alguns tipos de atributos especiais usados para demonstrar a estrutura das informações que eles representam – de modo a facilitar a busca dessas informações – ou o relacionamento entre as entidades. São eles:

1. Atributo composto: representa a estrutura das informações que serão armazenadas no atributo, por exemplo:

2. Atributo multivalorado: pode receber mais de um valor ao mesmo tempo. Um bom exemplo é o atributo habilidades de um funcionário, que será preenchido com a lista de suas aptidões separadas por vírgulas. Veja um exemplo de preenchimento: liderança, boa comunicação, bom relacionamento interpessoal. Assim, o atributo habilidades é considerado um atributo multivalorado.

3. Chave primária: atributo ou conjunto de atributos que identifica unicamente uma tupla (registro) em uma entidade. É expresso com um círculo preenchido, como mostra a figura 16.

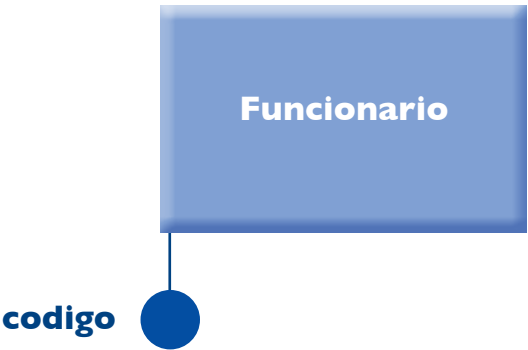



Figura 16
Chave primária.

4. Chave estrangeira: atributo que implementa o relacionamento entre entidades e permite o controle da integridade referencial, isto é, é um atributo que, fazendo parte da chave primária em uma entidade, é incluído em outra entidade ou relacionamento, implementando as ligações entre elas.

Relacionamento

É o elemento responsável por definir as características das ligações entre as entidades. Representado graficamente por um losango, seu nome é em geral expresso por um verbo ou uma locução verbal. Por exemplo a figura 17.

Figura 17
Relacionamento.

Auto-relacionamento: indica um relacionamento entre as ocorrências de um mesmo relacionamento. Para demonstrar melhor do que se trata, vale definir os papéis de cada um de seus lados, como mostra a figura 18.

Figura 18
Auto-relacionamento.

Cardinalidade

Serve para definir o tipo de relacionamento entre as entidades. Existem duas notações para identificar a cardinalidade. Uma delas refere-se simplesmente ao valor máximo que a cardinalidade daquele relacionamento pode alcançar, e é grafada com o número 1 (que representa 1 elemento da entidade) ou com a letra N (que representa muitos ou mais de um elemento da entidade). A outra expressa o número mínimo e o número máximo de ocorrências em um relacionamento. Neste caso, sua notação é (1:N), onde 1 representa o número mínimo e N o número máximo de ocorrências. São quatro as cardinalidades:

1. Relacionamentos 1 para N
Indicam que uma ocorrência na entidade A pode estar relacionada a N ocorrências da entidade B. No exemplo da figura 19 podemos verificar que um vendedor atende N clientes e que um cliente é atendido por apenas 1 vendedor.

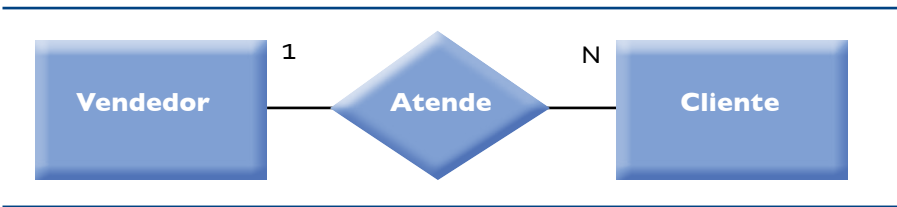
Figura 19
Relacionamento 1 para N.

2. Relacionamentos N para 1
Indicam que uma ocorrência na entidade B pode estar relacionada com N ocorrências na entidade A. No exemplo da figura 20, a 1 departamento podem pertencer N funcionários.

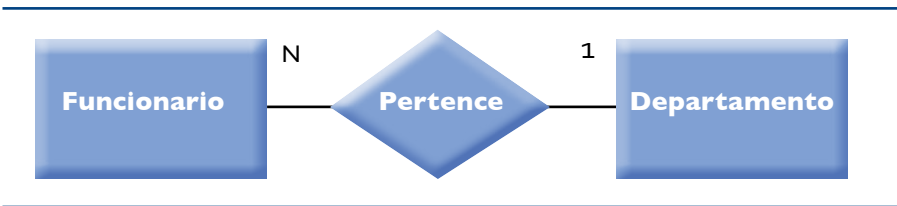
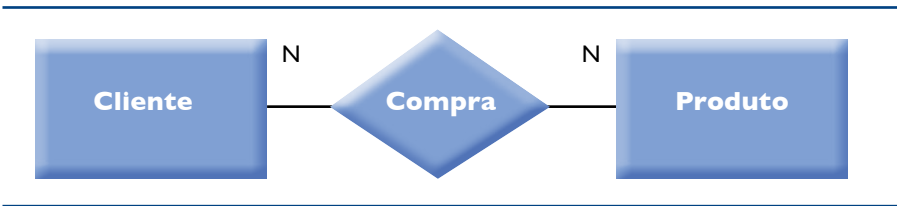
Figura 20
Relacionamento N para 1.

3. Relacionamentos N para N
Indicam que uma ocorrência na entidade A pode estar relacionada a N ocorrências na entidade B e que uma ocorrência na entidade B pode estar relacionada a N ocorrências na entidade A. No exemplo da figura 21 podemos observar que um cliente compra N produtos e que um produto pode ser comprado por N clientes.

Figura 21
Relacionamento N para N.

O relacionamento N para N possui uma característica especial. Também chamado de relacionamento com campos, sua implementação exige a inclusão de atributos (pelo menos na chave primária de cada uma das entidades envolvidas) e chave primária. Quando acontece a implementação do modelo físico, este relacionamento se torna uma tabela, como mostra a figura 22.

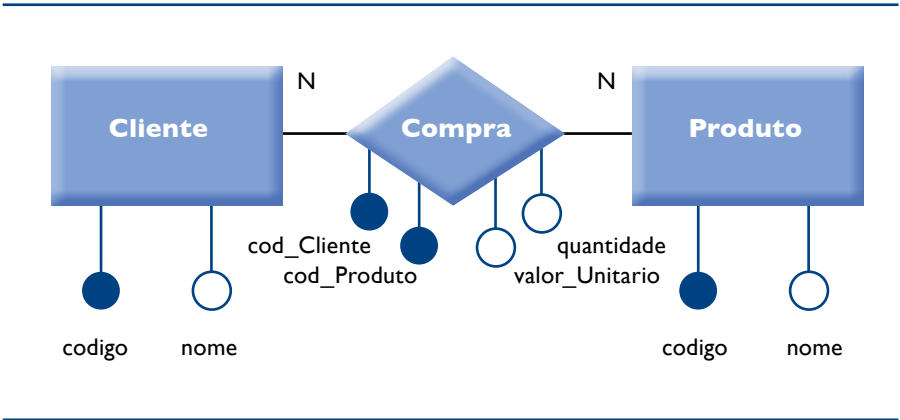



Figura 22
Relacionamento com campos.

Resumindo:

- Cliente tem os atributos "codigo" e "nome", "codigo" é a chave primária.
- Produto também tem os atributos "codigo" e "nome", tendo "codigo" como chave primária.
- Compra tem os atributos "cod_produto", "cod_cliente", que formam a chave primária, além dos atributos "quantidade" e "valor_unitario".

4. Relacionamentos 1 para 1

Indicam que uma ocorrência da entidade A pode estar relacionada a uma ocorrência na entidade B e que uma ocorrência da entidade B pode estar relacionada a uma ocorrência da entidade A (figura 23).

Figura 23
Relacionamentos 1 para 1.

Restrição

Muitas vezes, simplesmente definir um relacionamento não nos garante total entendimento da situação que ele se deve demonstrar, conforme a figura 24.

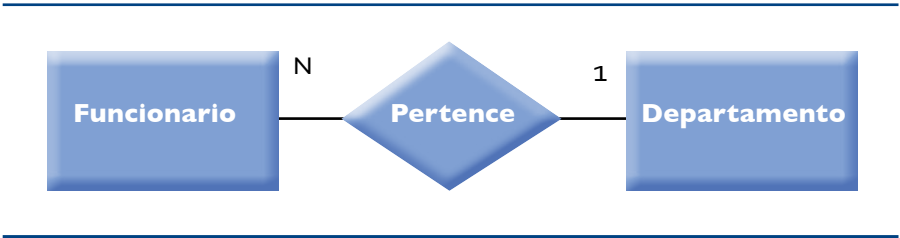
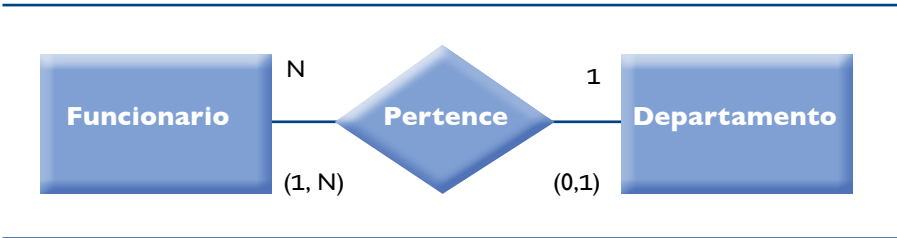



Figura 24
Relacionamento que não nos garante entendimento da situação.

Esse exemplo nos parece claro quanto ao relacionamento entre funcionário e departamento, mas e se perguntarmos: pode haver funcionário sem departamento associado?

Esse questionamento, num primeiro momento, pode parecer sem sentido, mas imagine uma situação em que os funcionários passam por treinamento e só são alocados em departamentos depois de serem avaliados. E, então, eles não são funcionários? Claro que são! E é para deixar mais clara esta definição que existe o conceito de restrição, que expressa quais são o maior e o menor valor do relacionamento. Para o caso proposto a notação ficaria como na figura 25:

Figura 25
Restrição.

Como se deduz que um funcionário pode pertencer a no mínimo nenhum e no máximo 1 departamento e 1 departamento possui no mínimo 1 e no máximo N funcionários. Essa representação nos ajuda a definir as restrições de integridade de nosso modelo e permite maior compreensão do banco de dados a ser construído. O que devemos entender é que pode haver funcionários sem departamento, mas não departamentos sem funcionário.

Agregação

Outro problema conceitual que é preciso definir é o relacionamento de uma entidade com um conjunto de entidades, isto é, esse relacionamento só existe se houver um conjunto de informações.

Imagine a seguinte situação:
Um cliente compra produtos e é atendido por um funcionário. Veja na figura 26 como exibir graficamente esse caso:

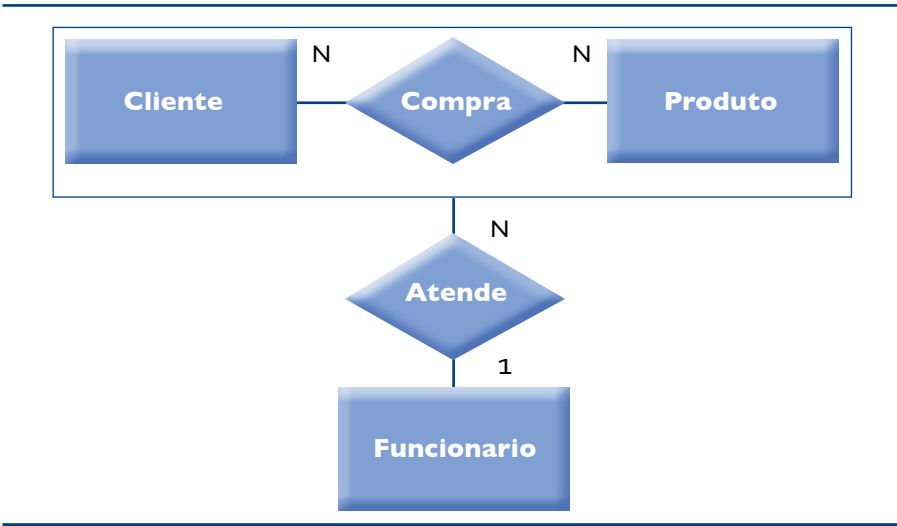
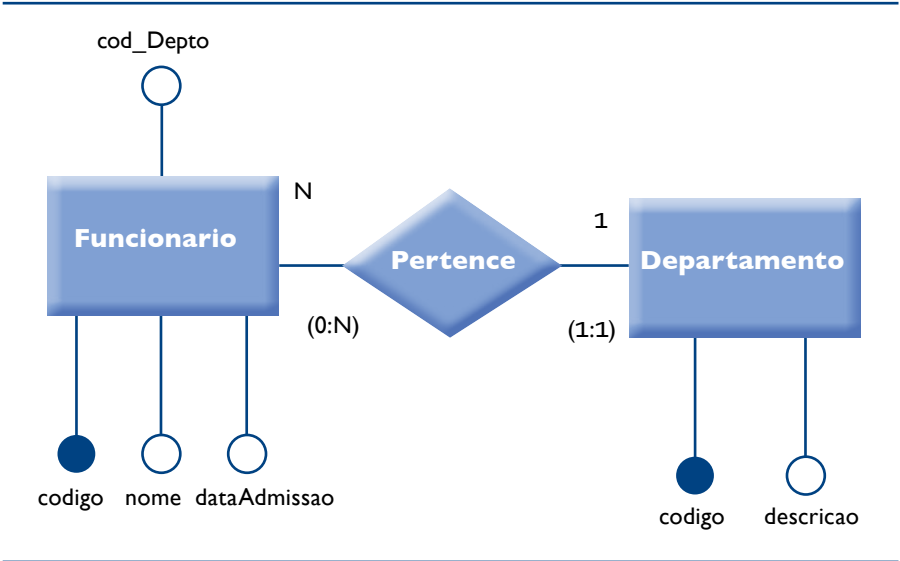
Figura 26
Agregação.

Interpretando o diagrama anterior, podemos dizer que um cliente compra N produtos e é atendido por 1 funcionário e que 1 funcionário atende N (clientes que compram produtos).

Diagrama de entidade e relacionamento

Parte do modelo de entidade e relacionamento, o diagrama é a representação gráfica dos elementos nele definidos. É montado após a denominação das entidades, dos seus atributos e relacionamentos (figura 27).

Figura 27
Diagrama de entidade e relacionamento.

É possível identificar nesse pequeno fragmento de diagrama quase todos os elementos do modelo visto até aqui:
Entidades: Funcionario e Departamento.
Relacionamento: Pertence.

Atributos:
- **da entidade Funcionario:** codigo, nome, dataAdmissao, cod_Depto.
- **da entidade Departamento:** codigo, descricao.

Chave primária:
- **da entidade Funcionario:** codigo.
- **da entidade Departamento:** codigo.

Chave estrangeira: o relacionamento Pertence com cardinalidade N para 1 faz com que seja necessário criar o campo cod_Depto na entidade Funcionario, que conterá o código do departamento a que cada funcionário pertence.

Restrições de integridade: podemos observar, pelas notações de restrição de integridade, que um funcionário tem de pertencer a um departamento. Já um departamento pode ser cadastrado sem um funcionário associado.

Vale, agora, propor um roteiro de passos para a elaboração do modelo de entidade e relacionamento e a criação do diagrama que o representará.

1. Listar as entidades candidatas a integrar o modelo.
2. Analisar e selecionar as entidades que realmente fazem parte do modelo, excluindo as demais.
3. Analisar os relacionamentos entre as entidades.
4. Definir a cardinalidade dos relacionamentos.
5. Definir os atributos das entidades e relacionamentos com campos e também as chaves primária e estrangeira (se houver).
6. Definir as restrições de integridade dos relacionamentos.
7. Desenhar o diagrama de entidade e relacionamento.

2.1.1. Estudo de caso

Vale imaginar como construir um modelo de entidade e relacionamento para o dono de uma pequena padaria, que será chamado de senhor João. No final, será elaborado o diagrama de entidade e relacionamento.

O senhor João vende, além de pães, vários outros tipos de produtos, tais como frios, laticínios, lanches, refrigerantes, sorvetes, balas, chicletes, chocolates, cartões telefônicos e artigos diversos expostos no balcão do caixa. Vende também, nos fins de semana, frango assado.

Na padaria, trabalham funcionários que executam as funções de caixa, atendente, auxiliar de limpeza e padeiro (Figura 28).

O senhor João quer que cada cliente receba um cartão com um código na entrada da padaria e que este cartão seja usado para registrar os produtos comprados pelos clientes. Os preços desses produtos deverão ser somados automaticamente assim que o cartão for entregue no caixa, que confirmará o valor total da compra, verificará a forma de pagamento escolhida, receberá o pagamento e, se for o caso, devolverá o troco ao cliente.

O senhor João também deseja controlar dos estoques para que não faltem produtos. Ele tem, portanto, necessidade de informações sobre:

- As vendas, isto é, precisa que sejam armazenados todos os dados de todas as vendas da padaria: quais produtos foram vendidos, em qual quantidade e

Figura 28
O senhor João tem necessidade de informações precisas sobre a sua padaria.

por qual valor, além de qual empregado registrou a venda e qual recebeu o pagamento.
- O estoque, de modo que cada produto vendido seja debitado no saldo.
- A necessidade de reposição de produtos – o modelo deve ter capacidade para gerar a qualquer momento a relação dos itens cujo saldo está abaixo do estoque mínimo desejável para facilitar a identificação daqueles que precisam ser repostos.
- A durabilidade e o uso dos cartões.
- Seus fornecedores – endereços, telefones e o nome do contato na empresa para efetuar a compra.

Uma observação importante: o senhor João já possui um controle fiscal e contábil de toda a movimentação, cujos documentos e registros ele envia semanalmente para seu contador. Fica, assim, para o modelo a ser proposto apenas o controle físico (estoque) e financeiro das transações.

Vamos pensar no problema proposto pelo senhor João, construir um modelo e demonstrá-lo por meio de um diagrama de entidade e relacionamento. Para isso, é preciso dar vários passos.

Passo 1

Listar as entidades candidatas a integrante do modelo.
Quando recebemos uma solicitação nesse formato, isto é, um texto que descreve a situação a ser tratada no modelo, uma prática bastante usada é no próprio texto fazer a identificação das possíveis entidades e relacionamentos, assim como dos principais atributos, grifando os substantivos e verbos essenciais para depois analisá-los a fim de atribuir-lhes os devidos papéis. Vejamos o que podemos selecionar em nosso texto.
No primeiro parágrafo temos os seguintes substantivos: senhor João, pães, produtos, frios, laticínios, lanches, refrigerantes, sorvetes, balas, chicletes, chocolates, cartões telefônicos, produtos, frango assado, balcão, caixa.
No segundo parágrafo encontramos: padaria, funcionários, funções, caixa, atendente, auxiliar de limpeza e padeiro.
No terceiro parágrafo podemos grifar: senhor João, clientes, cartão, código, produtos, padaria, caixa, valor total da compra, forma de pagamento, troco.
No quarto parágrafo identificamos: produto, senhor João, fornecedores.

Passo 2

Analisar e selecionar as entidades que realmente fazem parte do modelo, excluindo as demais.
Vamos avaliar os substantivos apenas uma vez, mesmo se eles aparecerem mais vezes, ou em mais de um parágrafo. Se forem exatamente iguais, será considerada a primeira análise.
Senhor João: é o dono da padaria e pode ser tratado como informação, pois também atende na padaria e trabalha no caixa. Não é entidade.
Pães, frios, laticínios, lanches, refrigerantes, sorvetes, balas, chicletes, chocolates, cartões telefônicos: tipos de produtos vendidos na padaria. São informações relacionadas à entidade produto, mas não são entidades.

Frango assado: é um produto vendido na padaria. Não é entidade.
Balcão: local físico onde ficam expostas as mercadorias. Não é informação.
Caixa: local físico na padaria no qual são registradas e pagas as compras. Não é informação.
Produtos: são os itens que o senhor João vende em sua padaria. Contêm um conjunto de atributos, tais como descrição, saldo e preço de venda. São uma entidade.
Padaria: é o tipo do estabelecimento que o senhor João possui. Tem um conjunto de atributos, como nome, endereço etc. É uma entidade.
Funcionários: são as pessoas que executam algum tipo de serviço necessário ao bom funcionamento da padaria. Possuem uma série de atributos que precisam ser armazenados para facilitar o controle e a consulta de suas informações. São uma entidade.
Funções: referem-se à qualificação do funcionário e ao tipo de serviço que ele exerce na padaria, logo, são atributos de funcionário.
Caixa, atendente, auxiliar de limpeza e padeiro: São os nomes das funções dos funcionários, informações que podem se relacionar com o atributo função.
Clientes: são os agentes de nosso modelo, aqueles que compram os produtos do senhor João. Possuem um conjunto de atributos tais como nome, endereço, telefone. Constituem uma entidade.
Cartão: é o item que representará o cliente na padaria. Demanda o controle de sua durabilidade e de seu uso. É associado ao registro das vendas e possui os atributos código, data de início de uso e data de fim de uso. É uma entidade.
Código: é um atributo da entidade cartão.
Valor total da compra: informação relevante da compra. E, assim, atributo da entidade compra.
Forma de pagamento: informação a opção de pagamento do cliente. É um atributo da entidade compra.
Troco: a diferença entre o valor da compra e o valor dado em dinheiro pelo cliente para o pagamento da compra. Atributo de compra.
Fornecedores: são os responsáveis por fornecer ao senhor João os produtos que ele vende. Possuem um conjunto de informações relevantes, como nome, endereço, telefone para contato. É uma entidade.

Assim, listamos como entidades: produtos, padaria, funcionários, clientes, compra, cartão e fornecedores. Como a boa prática manda nominar as entidades como substantivos no singular, teremos: produto, padaria, funcionário, cliente, cartão, forma de pagamento e fornecedor.

Analisando agora as **entidades** e pensando em sua relevância, temos que:
- A entidade padaria deve ser retirada de nosso modelo. Como a ideia é criar o modelo apenas para uma padaria, essa pode ficar fora de nosso escopo.

- A entidade cliente também pode ser retirada de nosso modelo, pois, entre as funcionalidades que o senhor João nos solicitou, não consta identificar o que cada cliente comprou. Você já se cadastrou em alguma padaria em que foi fazer compra? Na maioria dos casos, isso não acontece. Por isso, em nosso caso, o cliente será substituído pelo cartão.

Chegamos então a uma lista final de quatro entidades:
- **Produto**
- **Funcionário**
- **Cartão**
- **Fornecedor**

Passo 3

Analisar os relacionamentos entre as entidades.
Quanto aos relacionamentos entre as entidades listadas, identificamos:

- **Cartão compra Produto**
- **Funcionário atende (cartão compra Produto)**
- **Fornecedor Fornece Produto.**

Observe que identificamos anteriormente compra como sendo uma entidade e agora listamos-a como sendo um relacionamento. Por quê? Por se tratar de um relacionamento com campos que quando da criação do projeto físico torna-se uma tabela, assim como as entidades.

Passo 4

Definir a cardinalidade dos relacionamentos
Para os relacionamentos definidos, há as seguintes cardinalidades:

Cartão compra Produto
(Senhor Antonio entra na padaria para comprar um litro de leite. Recebe um cartão, vai ao balcão, pega o leite e 100 g de pão de queijo.)
Um cartão (o cartão que senhor Antonio pegou na entrada da padaria) compra vários produtos (um litro de leite e 100 g de pão de queijo) e um produto (leite) pode ser comprado por vários cartões (os da senhora Maria, do senhor Antonio, do Miro). Logo, a cardinalidade desse relacionamento é N para N, sendo necessário que se definam os atributos do relacionamento compra por se tratar de um relacionamento com campos.

Figura 29
Cartão compra produto.

Funcionário atende (cartão compra Produto)
Imagine a cena: a senhora Maria comprou dez pãezinhos e foi atendida pelo funcionário Laércio. Depois dela, Laércio atendeu o senhor Joaquim, Mariana, Pedro.

Figura 30
Funcionário atende.

Portanto, Laércio irá registrar dez pãezinhos no cartão da senhora Maria, depois registrará as compras do senhor Joaquim, Mariana, Pedro.

Logo, um funcionário (Laércio) atende vários (cartão compra produto – o cartão da senhora Maria x dez pãezinhos), mas um (cartão compra produto – o cartão da senhora Maria x dez pãezinhos) só é atendido (a cada vez que ela compra um produto na padaria) por um funcionário (Laércio), logo a cardinalidade deste relacionamento é N para 1.

Fornecedor fornece produto
O senhor Cardoso é dono de um atacado que vende bolachas e chocolates com o melhor preço da cidade. Toda vez que o dono da padaria, o senhor João, precisa comprar bolachas ou chocolates, ele liga para o senhor Cardoso e faz o pedido. No máximo até o dia seguinte o caminhão do senhor Cardoso para em frente à padaria e entrega as mercadorias solicitadas.

Logo, um fornecedor (senhor Cardoso) fornece vários produtos (bolachas e chocolates), e um produto (chocolate ao leite) pode ser vendido por vários fornecedores (senhor Cardoso, Maria Doceria, Bazar dos Amigos). Assim, a cardinalidade deste relacionamento é N para N.

É necessário que se definam os atributos do relacionamento fornece por se tratar de um relacionamento com campos.

Figura 31
Fornecedor.

Passo 5

Definir as restrições de integridade dos relacionamentos.
Ao identificar tais restrições de integridade, vamos também definir o valor mínimo e máximo de cada cardinalidade.

Cartão compra produto
As restrições de integridade são: um cartão pode comprar ao menos 0 (quando o cartão acabou de ser posto em uso) e no máximo N produtos (todos os produtos dos clientes que pegarem aquele cartão). Já um produto pode ser comprado por no mínimo 0 (o produto acabou de ser lançado e acabou de ser colocado na prateleira) e no máximo N cartões (todos os clientes da padaria).

Logo, as restrições de integridade são (0,N) e (0,N).

Funcionário atende (cartão compra Produto)
Um funcionário pode atender no mínimo 0 – o funcionário acaba de começar a trabalhar na padaria do senhor João – e no máximo N, Senhor Virgílio trabalha há quinze anos na padaria do senhor João. A senhora Maria acaba de chegar à padaria e Laércio, que está no balcão, vai atendê-la. Assim, o cliente pode ser atendido por no mínimo 1 e no máximo 1 funcionário. Logo, as restrições de integridade são (0,N) e (1,1).

Fornecedor fornece Produto
Um fornecedor fornece no mínimo 0 (Mário vende doces mas seus preços são sempre mais caros) e no máximo N produtos (senhor Cardoso). Já um produto (chocolate ao leite) é fornecido por no mínimo 1 e no máximo N fornecedores (senhor Cardoso, Maria Doceria, Bazar dos Amigos). Logo, as restrições de integridade são: (0,N) e (1,N).

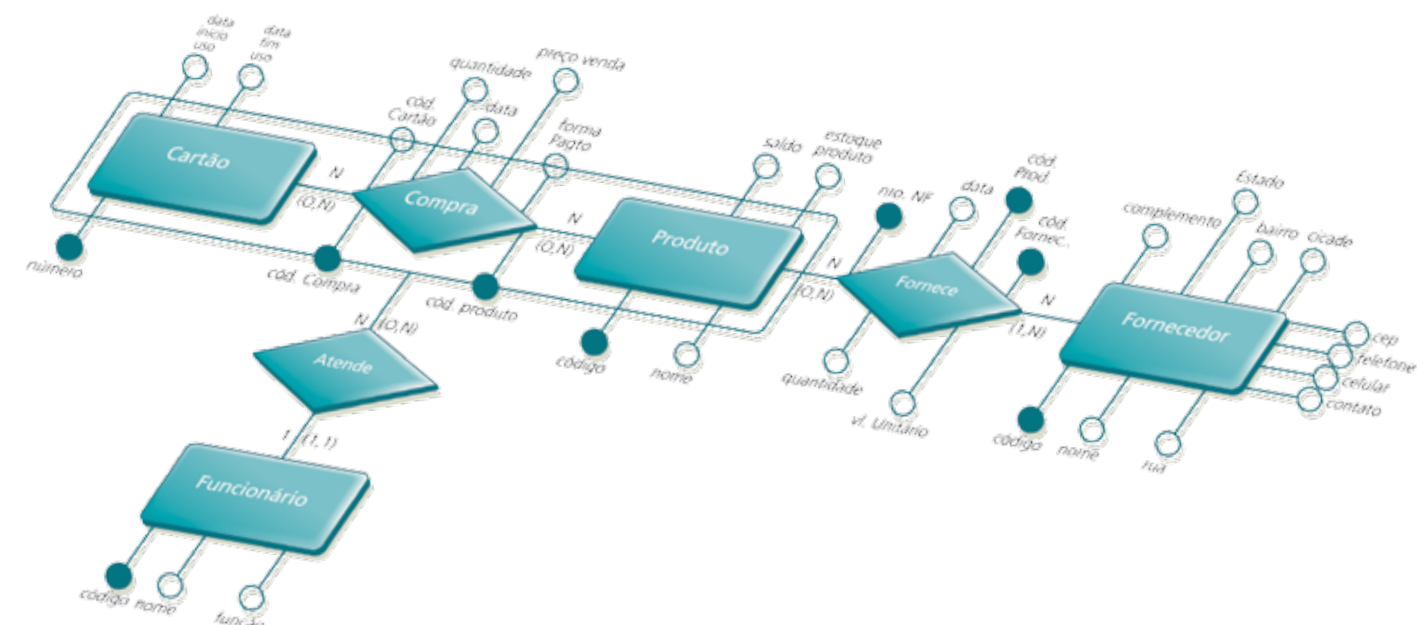
Passo 6

Definir os atributos das entidades e relacionamentos com campos e as chaves primária e estrangeira (se houver).
Para definir esses atributos temos de lembrar sempre do conceito de abstração, tentando selecionar apenas os aspectos relevantes para o escopo do problema em foco.

Entidades:
- **Cartao(codigo,data_inicio_uso,data_fim_uso):** o atributo código foi definido como chave primária, pois precisamos ter a certeza de que não existem dois cartões com o mesmo número e compartilharemos a responsabilidade dessa conferência com o sistema gerenciador de banco de dados.
- **Produto(codigo,nome,preco_venda,saldo,estoque_minimo):** o atributo código é chave primária, pois assim fica mais fácil fazer o controle para impedir que o valor deste atributo (codigo) se repita.
- **Funcionario(codigo,nome,funcao):** com o atributo codigo como chave primária.
- **Fornecedor(codigo,nome,rua,complemento,bairro,cidade,estado,cep, contato,telefone,celular):** o atributo codigo é chave primária.

Relacionamentos com campos.
Devemos nos lembrar de que um relacionamento com campos deve conter pelo menos as chaves primárias das entidades envolvidas. Eles podem conter outros atributos.
- **Compra(numero,data,forma_pagamento,codigo_produto,codigo_cartao, quantidade,valor_unitario,valor_total_compra).** Aqui as chaves primárias são os atributos numero, codigo_cartao e codigo_produto e as chaves estrangeiras os atributos codigo_cartao e codigo_produto.
- **Fornece(numero_nota,data,codigo_fornecedor,codigo_produto, quantidade, valor_unitario).** Nesse caso, as chaves primárias são os atributos numero_nota, codigo_fornecedor e codigo_produto e as chaves estrangeiras codigo_fornecedor e codigo_produto.

Passo 7 **Desenhar o diagrama de entidade e relacionamento**

Figura 32
Diagrama de entidade e relacionamento da padaria do senhor João.

2.2. Normalização

Normalização é um processo utilizado para acertar possíveis problemas estruturais das entidades e relacionamentos com campos criados – também chamados de anomalias – em um modelo de entidade e relacionamento. Consiste na análise dos atributos das entidades e relacionamentos com campos, sob o ponto de vista das regras chamadas formas normais, que descrevem, com base na teoria de conjuntos, na álgebra e no cálculo relacional, o que devemos ou não fazer nas estruturas das entidades e relacionamentos de nosso modelo, baseados em conceitos matemáticos.

Essa análise pode demonstrar a necessidade de alterarmos a estrutura de nossas entidades e relacionamentos com campos, dividindo ou agrupando seus atributos para aprimorar o processo de recuperação das informações (performance) e seu armazenamento, de modo a evitar perda, redundância e distorção da informação.

Sempre que formos obrigados pela aplicação das formas normais em nosso modelo a dividir entidades, temos que garantir que a divisão poderá ser revertida, isto é, que, mesmo particionada em duas ou mais entidades, uma entidade poderá voltar à sua formação original, por meio de operações de conjuntos.

Mas, antes de tratar das regras de normalização propriamente ditas, é necessário entender alguns conceitos da álgebra relacional, que serviram para definir se nossas entidades e relacionamentos estão ou não em uma forma normal.

Dependência funcional
Seja R uma relação e X e Y atributos de R. X e Y podem ser atributos simples ou compostos.

X → Y (o atributo X determina funcionalmente o atributo Y) sempre que duas tuplas quaisquer de R tiverem o mesmo valor para X, elas possuem também o mesmo valor para Y.

Exemplo:
Tendo a entidade funcionario os atributos codigo, nome, cidade e DDD, e sabendo que o codigo é a chave primária da entidade funcionario, se analisarmos esses atributos sob a óptica da dependência funcional, teremos:

codigo → nome
codigo → cidade
cidade → DDD

Logo, podemos dizer que os atributos nome e cidade dependem do atributo codigo. Já o atributo DDD depende do atributo cidade.

Definida a dependência funcional, podemos passar agora para a definição das formas normais.

Primeira Forma Normal (1NF)

Uma entidade está em Primeira Forma Normal, se e somente todos os seus atributos são atômicos, isto é, se contém um valor único (atômico) e não contém atributos multivalorados.

Exemplo:
Dada a entidade funcionario, definida com os atributos abaixo:

Funcionario(codigo,nome,data_admissao,data_demissao,habilidades)

Vemos que a entidade funcionario possui o campo multivalorado habilidades, o que não é permitido pela Primeira Forma Normal. Devemos então dividir a tabela funcionario de forma que o campo habilidades se torne uma nova entidade.

Então teremos:

Funcionario(codigo,nome,data_admissao,data_demissao)
Possui(cod_funcionario,cod_habilidade)
Habilidade(codigo,descricao)

Exemplo:

Fornecedor(codigo,nome,endereco,telefones)

Vemos que a entidade fornecedor tem como atributo composto endereco e como atributo multivalorado telefones.

Em relação ao atributo composto endereco, sabemos que o mesmo é composto, pois nele pressupõe-se incluir as informações de rua, complemento, bairro, cidade, estado e CEP. Ou substituímos o atributo endereco por seus atributos componentes (rua, complemento, bairro, cidade, estado e CEP) ou criamos uma outra entidade com o nome do atributo composto (endereco), tendo como atributos dessa nova entidade rua, complemento, bairro cidade, estado e CEP.

Já o campo telefones, por estar no plural, indica que nele poderá ser armazenado mais de um número. Pela regra, esse atributo precisa ser separado em outra entidade, que pode ser chamada de telefone e que conterá os diversos números de telefone do fornecedor.

Assim, temos:

Fornecedor(codigo,nome,rua,complemento,bairro,cidade,estado,cep)

Tendo como chave primária o atributo codigo.

Telefone(cod_fornecedor,nro telefone)

Tendo como chave primária os atributos cod_fornecedor e nro_telefone, já que isso garante que não será cadastrado o mesmo telefone para o forncedor e como chave estrangeira o atributo cod_fornecedor.

Fornecedor(codigo,nome)

Tendo como chave primária o atributo código.

Endereco(cod_fornecedor,rua,complemento,bairro,cidade,estado,cep).

Tendo como chave primária o atributo cod_fornecedor e como chave estrangeira o atributo cod_fornecedor.

Segunda Forma Normal (2NF)

Uma entidade encontra-se em Segunda Forma Normal se e somente estiver em Primeira Forma Normal e não tiver atributos com dependências parciais. No caso de uma chave primária composta, isto é, que possui mais de um atributo em sua composição, é denominada dependência parcial a dependência de um atributo não chave a apenas uma parte da **chave primária**.

Toda entidade que não possuir chave primária composta, isto é, com mais de um atributo, está em 2ª Forma Normal.

Tomemos como exemplo a tabela compra, descrita abaixo:

Compra(nro_nf,cod_fornecedor,cod_produto,data,nome produto, quantidade,valor_unitario,valor_total_nota)

Tendo como chave primária os atributos: nro_nf, cod_fornecedor, cod_produto
Se analisarmos a dependência funcional teremos:

nro_nf,cod_fornecedor → data
nro_nf,cod_produto → quantidade
nro_nf,cod_produto → valor_unitario
nro_nf,cod_fornecedor → valor_total_nota
cod_produto → nome_produto

Vemos agora que nem todos os atributos dependem da chave primária. O que não é permitido pela Segunda Forma Normal. Para que essa entidade fique em 2NF, teremos de desmembrá-la.

Ela ficará assim:

Compra(nro_nf,cod_fornecedor,data,valor_total_nota)

Tendo como chave primária os atributos nro_nf cod_fornecedor e como chave estrangeira o atributo cod_fornecedor.

Item_compra(nro_nf,cod_produto,quantidade,vl_unitario)

Tendo como chave primária os atributos nro_nf e cod_produto e como chaves estrangeiras os atributos nro_nf e cod_produto.

Produto(codigo,nome)

Tendo como chave primária o atributo codigo.

Terceira Forma Normal (3NF)

Uma entidade está em Terceira Forma Normal se e somente estiver em Primeira e em Segunda Forma Normal e todos os atributos não chave dependerem funcionalmente da chave primária.

Exemplo:

Pedido(nro_pedido,data,cod_cliente,nome_cliente,email_cliente,valor_total_pedido)

Vamos verificar a dependência funcional dos atributos:

nro_pedido → data
nro_pedido → cod_cliente
nro_pedido → valor_total_pedido
cod_cliente → nome_cliente
cod_cliente → email_cliente

Verificamos que os atributos nome_cliente e email_cliente não são dependentes da chave primária e sim do atributo cod_cliente. Será necessário então desmembrar a entidade pedido.

Pedido(nro_pedido,data,cod_cliente,valor_total_pedido)

Que terá como chave primária o atributo nro_pedido.

Cliente(cod_cliente,nome_cliente,email_cliente)

Que terá como chave primária o atributo cod_cliente.

Forma Normal Boyce-Codd (BCNF)

Uma entidade está em BCNF se e somente estiver em Terceira Forma Normal e todos os atributos não chave dependerem apenas da chave primária.

Exemplo:

Cliente(cod_cliente,nome_cliente,email_cliente)

Que terá como chave primária o atributo cod_cliente.

cod_cliente → nome_cliente
cod_cliente → email_cliente

Todos os atributos não chave dependem funcionalmente apenas da chave primária. Logo, está em BCNF.

Vamos agora, como mais um exemplo de aplicação das regras de normalização, aplicar a normalização nas entidades do modelo da padaria do senhor João.

Ao final da montagem de nosso modelo ficamos com as seguintes entidades:

Cartao(codigo,data_inicio_uso,data_fim_uso). A chave primária foi definida como sendo o atributo codigo.

Produto(codigo,nome,preco_venda,saldo,estoque_minimo), tendo o atributo codigo como chave primária.

Funcionario(codigo,nome,funcao), tendo o atributo codigo como chave primária.

Fornecedor(codigo,nome,rua,complemento,bairro,cidade,estado,cep,contato,telefone,celular), tendo o atributo codigo como chave primária.

Compra(numero,data,forma_pagto,codigo_produto,codigo_cartao, quantidade,valor_unitario,valor_total_compra), tendo como chave primária os atributos numero, codigo_cartao e codigo_produto e como chaves estrangeiras os atributos codigo_cartao e codigo_produto.

Fornece(numero_nota,data,codigo_fornecedor,codigo_produto, quantidade, valor_unitario), tendo como chave primária os atributos numero_nota, codigo_fornecedor e codigo_produto e como chaves estrangeiras codigo_fornecedor e codigo_produto.

Cartao(codigo,data_inicio_uso,data_fim_uso) – a chave primária foi definida sendo o atributo codigo.

Agora vamos analisar cada uma das entidades.

OBSERVAÇÕES

• Sempre que for preciso desmembrar uma entidade, isso deve ser feito de maneira que seja possível retornar à forma anterior, evitando, com isso, a perda de informações.
• Sempre que se fizer o desmembramento de uma entidade, as tabelas resultantes devem ser submetidas novamente às formas normais para se ter certeza de que estão corretamente normalizadas.

Entidade cartão

Cartao(codigo,data_inicio_uso,data_fim_uso)

O atributo código foi definido como chave primária.
Está em Primeira Forma Normal, pois todos os seus atributos são atômicos.
A entidade cartao está em Segunda Forma Normal, pois sua chave primária não é composta.

Vamos verificar a dependência funcional da entidade cartao:

codigo→data_inicio_uso
codigo→data_fim_uso

Logo, a entidade cartao está em Terceira Forma Normal, pois todos os atributos são dependentes funcionalmente da chave primária.

A entidade cartao está na Forma Normal Boyce-Codd, pois todos os seus atributos não chave dependem unicamente da chave primária.

Cartao(codigo,data_inicio_uso,data_fim_uso)

Tendo o atributo codigo como chave primária.

Entidade Produto

Produto(codigo,nome,preco_venda,saldo,estoque_minimo)

Tendo o campo codigo como chave primária.

Está em Primeira Forma Normal, pois ela não possui atributos multivalorados e atributos compostos.

A entidade produto está em Segunda Forma Normal, pois sua chave primária não é composta. Analisando sua dependência funcional, temos:

codigo →nome
codigo →preco_venda
codigo →saldo
codigo →estoque_minimo

A entidade produto está em Terceira Forma Normal, pois todos os atributos não chave dependem funcionalmente da chave primária. A entidade produto está na Forma Normal Boyce-Codd, pois todos os atributos não chave são dependentes apenas da chave primária.

Entidade funcionario

Funcionario(codigo,nome,funcao)

Tendo o atributo codigo como chave primária.

Está em Primeira Forma Normal, pois não possui atributos multivalorados nem atributos calculados.

A entidade funcionario está em Segunda Forma Normal, pois não possui chave primária composta.

Verifiquemos a dependência funcional da entidade funcionario:

codigo →nome
codigo →funcao

A entidade funcionario está em Terceira Forma Normal, pois todos os seus atributos não chave são dependentes da chave primária.

A entidade funcionario está na Forma Normal Boyce-Codd, pois todos os atributos não chave dependem apenas da chave primária.

Funcionario(codigo,nome,funcao)

Tendo o campo codigo como chave primária

Entidade Fornecedor

Fornecedor(codigo,nome,rua,complemento,bairro,cidade,estado,cep, contato,telefone,celular)

Tendo o atributo codigo como chave primária.

Está em Primeira Forma Normal, pois não possui atributos multivalorados nem atributos calculados.

A entidade fornecedor está em Segunda Forma Normal, pois sua chave primária é simples.

Análise da dependência funcional da entidade fornecedor.

codigo → nome
codigo → rua
codigo → complemento
codigo → bairro
codigo → cidade
codigo → estado
codigo → cep
codigo → contato
codigo →telefone
codigo →celular

- A entidade fornecedor está em Terceira Forma Normal, pois todos os atributos não chave dependem funcionalmente da chave primária.
- A entidade fornecedor está na Forma Normal Boyce-Cood, pois todos os seus atributos não chave dependem apenas da chave primária.

Entidade compra

Compra(numero,data,forma_Pagto,codigo_produto,codigo_cartao,quantidade,valor_unitario,valor_total_compra)

Tendo como chave primária os atributos numero, codigo_cartao e codigo_produto e como chaves estrangeiras os atributos codigo_cartao e codigo_produto.

Está em Primeira Forma Normal, pois todos os seus atributos são atômicos.

Verificando a dependência funcional da entidade compra, temos:

numero,codigo_cartao,cod_produto → data
numero,codigo_cartao,cod_produto → forma_Pagto
numero,cod_produto → quantidade
numero,cod_produto → valor_unitario

Nem todos os atributos dependem da chave primária, então, para deixar a entidade compra em Segunda Forma Normal, devemos desmembrá-la (compra e ItemCompra), assim:

Compra (numero, cod_cartao,data,valor_total_compra,forma_pagto)

E com a chave primária contendo o atributo numero e a chave estrangeira cod_cartao.

ItemCompra(numero,cod_produto,quantidade,valor_unitario)

Com chave primária contendo os atributos numero e cod_produto e como chaves estrangeiras os atributos numero e cod_produto.

- A entidade Compra encontra-se em Segunda Forma Normal.
- A entidade ItemCompra encontra-se em Segunda Forma Normal.
- A entidade Compra encontra-se em Terceira Forma Normal, pois como vimos na verificação da dependência funcional, todos os atributos não chave dependem da chave.
- A entidade Compra está na Forma Normal Boyce-Codd, pois todos os atributos não chave dependem unicamente da chave primária.
- A entidade ItemCompra está na forma normal Boyce-Codd, pois todos os atributos não chave dependem unicamente da chave primária.

Entidade fornece

Fornece (numero_nota, data, codigo_fornecedor,codigo_produto, quantidade, valor_unitario)

Tendo como chave primária os atributos numero_nota, codigo_fornecedor e codigo_produto e como chaves estrangeiras codigo_fornecedor e codigo_produto.

A entidade Fornece está em Primeira Forma Normal, pois todos os seus atributos são atômicos. Vale verificar a dependência funcional dessa entidade.

numero_nota,cod_fornecedor,cod_produto → data
numero_nota,cod_fornecedor,cod_produto → valor_total
numero_nota,cod_produto → quantidade
numero_nota,cod_produto → valor_unitario

Nem todos os atributos não chave dependem completamente da chave, logo é preciso desmembrar seus atributos:

Fornece(numero_nota,cod_fornecedor, data,valor_total)

Ficando como chave primária os atributos numero_nota e cod_fornecedor e como chave estrangeira o atributo cod_fornecedor.

ItemFornece(numero_nota,cod_produto,quantidade,valor_unitario)

Tendo como chave primária os atributos numero_nota e cod_produto e como chaves estrangeiras os atributos numero_nota e cod_produto.

- A entidade Fornece está em Segunda Forma Normal, pois todos os atributos não chave dependem completamente da chave.
- A entidade ItemFornece está em Segunda Forma Normal, pois todos os atributos não chave dependem completamente da chave.
- A entidade Fornece está em Terceira Forma Normal, pois todos os seu atributos não chave dependem da chave primária
- A entidade ItemFornece está em Terceira Forma Normal, pois todos os seus atributos não chave dependem da chave primária.
- A entidade Fornece está na Forma Normal Boyce-Codd, pois todos os atributos não chave dependem unicamente da chave primária.
- A entidade ItemFornece está na Forma Normal Boyce-Codd, pois todos os atributos não chave dependem unicamente da chave primária.

Há outros passos a serem seguidos até o nosso modelo ser implementado, mas, só para entendermos o que as entidades representam, vejamos como ficarão as entidades da padaria do senhor João depois de implementadas. Em nossa representação tabela *Compra*, a primeira linha é o nome da tabela (entidade), a segunda linha os nomes dos campos (atributos) e a partir da terceira linha são as informações armazenadas que chamamos de registros ou tuplas. Veja como os relacionamentos permitem entender perfeitamente as informações gravadas nas tabelas.

COMPRA

numero	cod_cartao	data	forma_pagto	valor_total compra
132	3	21/05/2009	dinheiro	10,60
133	2	21/05/2009	dinheiro	13,60
134	6	23/05/2009	cartao	52,20

Vamos analisar, por exemplo, a tabela ItemCompra. Veja que a soma dos itens da compra de código 132 (campos quantidade *valor_unitario) é o mesmo valor gravado no campo valor_total_compra da tabela compra. Isto é integridade de dados. Aproveite para olhar detalhadamente para as demais tabelas, para entender como os atributos se transformam em campos e como esses campos se relacionam, permitindo acesso rápido e eficiente às informações. Olhe agora para as tabelas Fornecedor, Fornece e ItemFornece. Só de olhar, já sabemos que, no dia 21/05/2009, o senhor João comprou do senhor Pedro Parente 120 litros de leite B e dois achocolatados em pó.

ITEMCOMPRA

numero_nota	cod_produto	quantidade	valor_unitario
132	2	2	3,20
132	3	1	4,20

Como sabemos disso? Veja o caminho que fazemos saindo de Fornece indo para ItemFornece; veja a implementação do relacionamento, pois os campos numero_nota das duas tabelas tem o valor 1 e o código do produto de ItemFornece equivale aos produtos leite B e achocolatado em pó. Vemos agora na prática o que definimos na teoria. Preste atenção nas outras tabelas e veja se você acha mais informações interessantes. Olhe também a importância dos atributos chave primária e chave estrangeira em nossa implementação. Ainda há muito o que aprender para que os modelos se transformem em sistemas de qualidade, mas agora já se sabe como os modelos se transformam em banco de dados de aplicações.

FORNECEDOR

Codigo	Nome	Rua	Complemento	Bairro	Cidade	Estado	CEP	Contato	Telefone	Celular
1	Cardoso Alimentos	Rua 2 nº 32		Jd. Boa Esperança	Campinas	SP	12123-123	Sr. Cardoso	99-3234-0000	99-9999-0000
2	Maria Doceria	Rua 34 nº 123	Atrás da Igreja	Centro	Hortolândia	SP	12123-123	D. Maria		
3	Pedro Parede	Rua 4 nº 120		Jd. Cachoeira	Caldas	MG	04321-789	Sr. Pedro	99-9999-8976	87-9999-0000

FORNECE

numero_nota	cod_fornecedor	data	valor_total
1	3	21/05/2009	123,12
2	2	21/05/2009	34,20
4	6	23/05/2009	48,90

ITEMFORNECE

numero_nota	cod_produto	quantidade	valor_unitario
1	2	120	1,00
1	3	2	1,56

PRODUTO

Codigo	Nome	Preço_Venda	Saldo	estoque_Minimo
2	Leite B.	1,89	32	4
3	Achocolatado em pó	3,45	13	2
15	Leite condensado	2,11	54	12

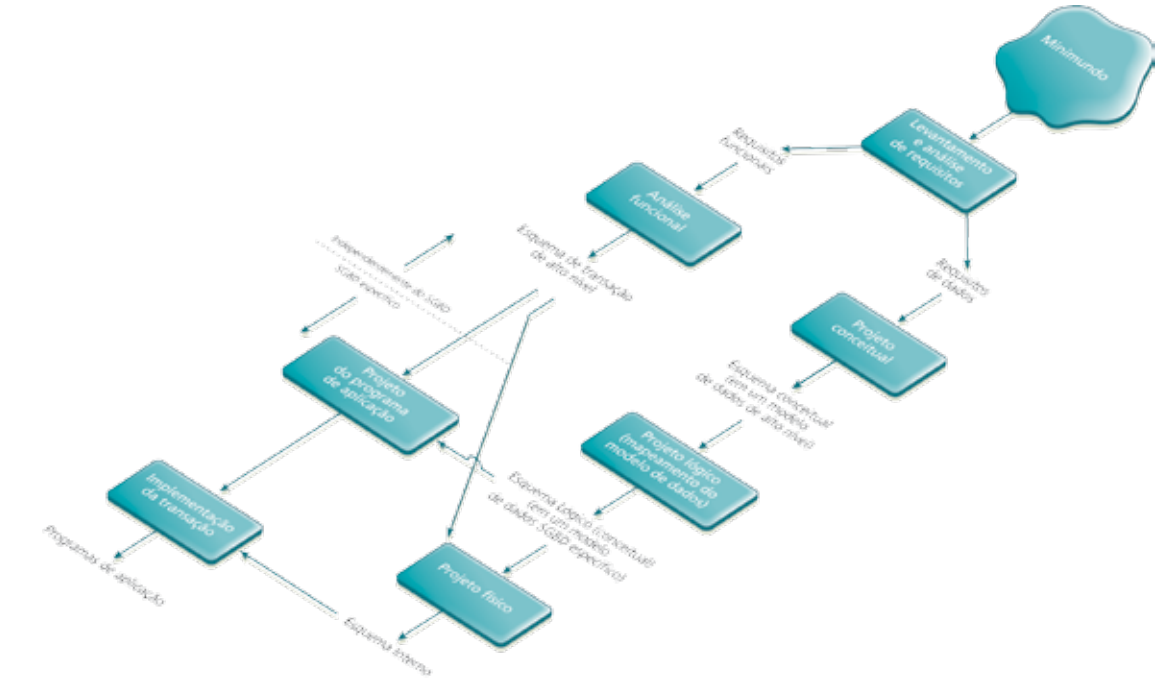
FUNCIONARIO

Codigo	Nome	Função
1	Sr. João	Dono
2	Laercio da Silva	Padeiro
3	Maria padilha	Atendente

2.3. Fases de um projeto utilizando o modelo ER

No caso fictício da padaria do senhor João, descobrimos, por meio de um relato, como a padaria funcionava e quais eram as expectativas de seu dono em relação a como passaria a operar e ao tipo de informação que o novo sistema lhe permitiria obter. Na vida real, dificilmente acontece aquela sequência de ações idealizadas para compor o exemplo. Geralmente, o usuário (cliente) não sabe muito bem do que precisa, quando decide informatizar algum processo do seu negócio. Nós é que devemos nos aproximar dele para reunir elementos que permitam desenvolver a solução que o atenda da melhor forma. É preciso também oferecê-la de acordo com o escopo esperado, isto é, construída dentro do prazo solicitado e com o orçamento disponível. Sim, porque para viabilizar um projeto essa dualidade é fundamental: a correta conjunção de tempo combinado e cujo valor esteja dentro do investimento que o cliente está disposto a fazer.

Não é, portanto, uma tarefa fácil. Corre-se o risco da perda de foco e de ritmo de trabalho, se não forem usadas técnicas que sinalizem corretamente e facilitem nosso caminho. Felizmente, há um roteiro para a criação de soluções informatizadas que utilizam o Modelo de Entidade e Relacionamento, um guia que pode nos conduzir a um bom resultado final, ou seja, o projeto pronto e instalado na empresa do cliente (figura 33).

Figura 33
Roteiro para criar uma solução informatizada.

O roteiro oferece uma série de indicações a seguir até que se alcance o produto final – o software implementado. Há, porém, algumas hipóteses que podem alterar os rumos do projeto. Durante seu desenvolvimento, podemos concluir, por exemplo, que a solução inicialmente imaginada não é econômica ou tecnicamente viável; ficará cara demais, sem proporcionar o resultado esperado, no prazo estabelecido. Pode-se até concluir que não há necessidade de uma solução informatizada para o problema proposto, sugerindo que seja resolvido apenas por meio de uma simples mudança de procedimento (inclusão e/ou alteração de ações dos usuários no processo).

Vale, então, conhecer e analisar em detalhes cada um dos passos sugeridos no roteiro, para então aplicá-los ao nosso projeto para a padaria do senhor João. Verificaremos, assim, quais pontos devem ser levados em conta em cada passo e qual será o produto final para cada fase proposta. Comecemos pelo minimundo.

2.3.1. Minimundo

O minimundo – geralmente o ponto inicial do trabalho – é a parte do mundo real que é o foco do projeto ou de nossa análise. No nosso caso, é a padaria do senhor João. Usaremos as técnicas descritas no capítulo 1, para conhecer melhor o funcionamento da empresa, em especial o foco do problema, de acordo com os pontos levantados pelo senhor João: a dinâmica do atendimento ao público, o processo de compra e venda de mercadorias e a necessidade de reposição de estoques no prazo desejável.

2.3.2. Levantamento de requisitos

Por meio das técnicas que estudamos no capítulo anterior, vamos levantar os requisitos para a solução do problema proposto. Avaliaremos, portanto, o tamanho do problema, o que se espera da solução, quem são os usuários chave, quais processos estão envolvidos e o que cada um desses processos oferece de informação relevante à solução que estamos buscando.

Quem pode nos dar essas respostas são os usuários e seus procedimentos. Lembre-se de que devemos escolher uma ou mais técnicas descritas para conhecer bem o problema.

DICA

Desde o início, as informações levantadas no cliente devem ser registradas para que se tornem base de nosso entendimento do problema e referência para consulta posteriores.

De início, podemos recorrer aos métodos de entrevista para obter informações do senhor João e seus funcionários com o objetivo de compreender o funcionamento da padaria. Também podemos observar os funcionários em seu dia-a-dia para verificar a dinâmica do trabalho, suas rotinas, particularidades e informações geradas nas diferentes operações. Para cada entrevista, procedimento ou etapa é necessário produzir uma documentação, isto é, anotar de forma clara e isenta as informações obtidas e suas fontes (quem nos deu tal informação, como chegamos a determinada conclusão). Este procedimento nos permitirá construir uma base de apoio que poderemos consultar sempre que surgir alguma dúvida em relação ao processo ou à solução imaginada. Cada passo também pode nos apontar novas informações a analisar ou procedimentos a seguir, além de meios de sanar dúvidas que eventualmente surjam no caminho e indicações sobre quem pode nos ajudar a dirimi-las. É fundamental, ainda, cultivar a melhor relação possível com as pessoas envolvidas no processo, deixando clara sua importância para o trabalho, mesmo depois que tivermos encerrado a fase de levantamento de requisitos.

2.3.3. Análise de requisitos

Entendida a dinâmica de funcionamento de nosso minimundo e obtidas as informações relevantes ao foco da solução imaginada, é hora de analisar essas informações, separá-las e classificá-las de forma podermos continuar a desenvolvê-la, verificando inclusive se a melhor opção é mesmo informatizar, e, em caso positivo, se haverá mesmo condições de satisfazer as necessidades do usuário no prazo previsto. O primeiro passo, agora, é separar os requisitos levantados e classificá-los em requisitos de dados e requisitos funcionais. Mas, antes de tudo, resta esclarecer bem quais são esses requisitos.

DICA

Lembre-se: podem surgir dúvidas em qualquer fase do desenvolvimento do projeto e você precisará de mais informações e opiniões do usuário até o fim do processo.

2.3.4. Requisitos de dados

Trata-se, aqui, de toda e qualquer informação relevante para a solução em análise, tanto as geradas quanto as consultadas com o objetivo de concluir determinada tarefa. Por exemplo, a quantidade e o valor do produto comprado, ou seja, o montante do pagamento, são requisitos de dados que devem ser classificados para que se possa construir um modelo de dados.

2.3.5. Requisitos funcionais

São aqueles que descrevem funcionalidades e serviços do sistema. Tal definição dependerá do tipo do software, dos resultados esperados e do tipo do sistema em que o software será aplicado. Exemplos: o sistema deve oferecer diversas maneiras de visualizar os dados, de acordo com o perfil do usuário; os relatórios

Figura 34
As várias operações do negócio padaria.

têm de ficar disponíveis para impressão, de acordo com o nível hierárquico de cada um deles. Para saber qual é esse nível, é necessário que ele se identifique no sistema, digamos, por meio de uma rotina de login, em que ele se apresente via senha, geralmente criada por ele mesmo.

Podemos considerar requisitos funcionais também a manutenção, isto é, inclusão/alteração/exclusão e consulta de todas as entidades identificadas na solução.

Observe que narramos a situação atual da padaria do senhor João, isto é, sem a solução informatizada, e a submetemos às regras de requisitos que definimos no capítulo anterior (necessário, não-ambíguo, verificável, conciso, alcançável, completo, consistente, ordenável e aceito). Agora, avaliemos as operações envolvidas nas vendas (figura 34).

Situação
A senhora Maria vai até o balcão de pães e solicita dez pãezinhos ao funcionário Laércio, que está atendendo naquele momento.
Laércio pega o saco de papel adequado a tal quantidade, vai até a cesta de pães e com o pegador coloca no saco os dez pãezinhos. Em seguida fecha o saco, coloca-o sobre a balança e digita, no equipamento, o preço do quilo do pão. Toma então um pedaço de papel que está sobre o balcão no qual anota o peso dos pães e o valor apresentado na balança, além da palavra pão, e entrega o pacote e o pedaço de papel à senhora Maria, perguntando-lhe se vai querer mais alguma coisa.

Como separar requisitos de dados de requisitos funcionais nesta situação ?
Primeiro, vamos tentar simplificar o problema, selecionando apenas as informações relevantes. Para isso, vamos reconstruir quadro a quadro a situação, formulando algumas perguntas:

1 - A senhora Maria vai até o balcão de pães e solicita dez pãezinhos ao funcionário Laércio, que está no balcão neste momento.
Quais são os requisitos importantes para nossa solução nesta frase?

A senhora Maria se deslocar até o balcão de pães não é um requisito relevante, pois não nos interessa, no momento, como ela chegou ao balcão e sim o que fez ao chegar lá.

---

2 - Solicita dez pãezinhos ao funcionário Laércio que está atendendo naquele momento.
Ou seja, a senhora Maria (a cliente) solicita dez pãezinhos (dez unidades do produto pãozinho) para o funcionário Laércio que está no balcão.

Concluímos, aqui, que o cliente solicita uma certa quantidade de um determinado produto a certo funcionário.

Laércio ser o funcionário que está no balcão no momento, e a senhora Maria a cliente, são informações que nos demostram apenas que há um funcionário e um cliente envolvidos na ação.

Continuando nossa análise:

3 - Laércio pega o saco de papel adequado a tal quantidade, vai até a cesta de pães e com o pegador coloca no saco os dez pãezinhos. Em seguida, fecha o saco, coloca-o sobre a balança e digita no equipamento o preço do quilo do pão.

Temos de importante aqui que o funcionário pesa o produto, digitando o preço do quilo na balança.

---

4 - Toma, então, um pedaço de papel que está sobre o balcão no qual anota o peso dos pães e o valor apresentado na balança, além da palavra pão; entrega o pacote e o pedaço de papel à senhora Maria, perguntando-lhe se vai querer mais alguma coisa.

Aqui temos que:
funcionário anota a quantidade, o tipo de produto e seu preço em um pedaço de papel e o entrega ao cliente.

Em resumo:

- O cliente solicita uma quantidade de um certo produto ao funcionário.
- O funcionário pesa o produto, digitando o preço do quilo na balança.
- O funcionário anota a quantidade, o produto e seu preço em um pedaço de papel e o entrega ao cliente.

Antes de separarmos os requisitos funcionais dos requisitos de dados, precisamos pensar se as três ações descritas no resumo fazem parte do escopo de nosso projeto. Vejamos.

O cliente pedir o produto e o empregado separá-lo não são fatos relevantes na situação, pois são ações que não serão alteradas: o cliente continuará a pedir os produtos aos funcionários e os procedimentos a seguir continuarão sendo os mesmos.

O que importa, então, é que o funcionário anota a quantidade, o produto e seu preço em um pedaço de papel e o entrega ao cliente.

Aí temos uma ação que será modificada por nossa solução, pois sabemos que o senhor João quer que o cliente entregue um cartão ao funcionário, que nele registrará as compras e o devolverá ao cliente.

Os requisitos de dados contidos nesta situação são: funcionário, quantidade de produto, valor total de produto, nome do produto e do cliente. Já o requisito funcional é: anota, pois o funcionário deverá anotar os itens comprados pelo cliente.

Requisitos colhidos até agora.
Ao terminar a análise dessa parte do problema, teremos:

- Requisitos de dados: funcionário, quantidade comprada, valor total do produto e descrição do produto.
- Requisitos funcionais: funcionário anota itens solicitados pelo cliente, isto é, o funcionário deverá ter um "lugar" no sistema para anotar as compras do cliente.

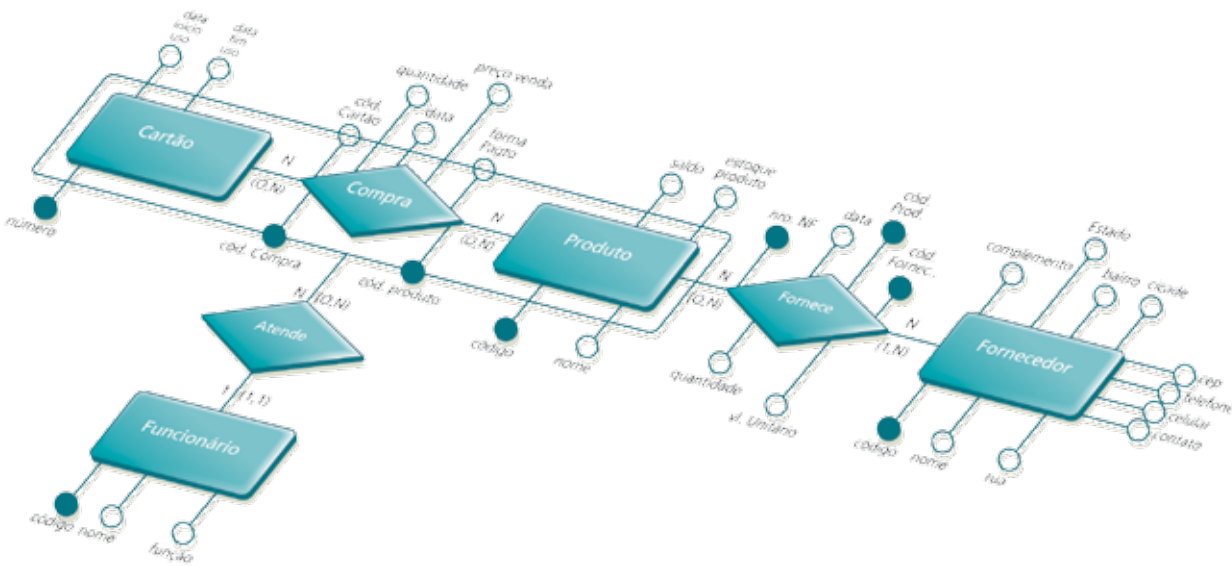
2.3.6. Projeto conceitual

Depois de analisar e classificar todos os requisitos levantados, devemos nos concentrar nos requisitos de dados, agrupando-os em entidades e relacionamentos. Ao pensarmos nas entidades, devemos verificar quais são as informações relevantes em vários aspectos, seguindo os sete passos propostos no tópico Modelo de entidade e relacionamento, estudado no início deste capítulo, para montar o diagrama de entidade e relacionamento, que vai retratar nosso modelo conceitual e classificará as entidades geradas segundo as regras de normalização. Nosso projeto conceitual ficará como sugere a figura 35.

2.3.7. Projeto lógico

Com o projeto conceitual montado, precisamos, agora, definir o Sistema Gerenciador de Banco de Dados (SGBD) que empregaremos para implementar nossa solução de software. Existem inúmeras opções de SGBD no mercado, em uma gradação de valor que vai dos gratuitos aos bastante caros. Todos implicam vantagens e desvantagens, que você deverá analisar juntamente com os demais participantes do projeto.

Figura 35
A aplicação do diagrama de entidade e relacionamento, na prática.

Diagrama de entidade e relacionamento da padaria do sr. João

Depois de escolher o SGBD, devemos verificar quais são os tipos de dados que este aceita, bem como seus tamanhos, pois tais características variam muito. Existem quatro tipos básicos de dados: texto, número, data e hora. Para a padaria do senhor João, usaremos o Banco de dados MySQL. Com essa definição, podemos passar à etapa seguinte: fazer o projeto lógico das entidades e relacionamentos com os campos que usaremos na solução. É preciso definir os atributos de cada entidade, além de tipo, tamanho e obrigatoriedade ou não de cada atributo, e, ainda, avaliar se o atributo é chave primária ou estrangeira e se é único. Resta conhecer o significado das classificações dos atributos (Leia o quadro abaixo).

SIGNIFICADOS E DICAS	
Nome	palavra que indique o atributo no contexto da entidade; deve-se evitar, o uso de abreviações e números.
Tipo	indica a forma como o atributo será armazenado (data, texto, número, etc.).
Tamanho	expressa o número de caracteres que o atributo ocupará na entidade. É preciso cuidado para definir o tamanho do atributo para que não se crie problemas difíceis de resolver no futuro, pois alterar o tamanho de um atributo, depois que o sistema está em produção, implica alterar todas as telas e relatórios nos quais tal atributo aparece. Por exemplo, se criarmos o campo código da tabela cliente com 3 posições, poderão ser cadastrados no máximo 999 clientes, o que representará uma limitação no sistema.
Obrigatório	define se este atributo tem de ser preenchido para que se possa incluir uma informação nesta entidade ou não.
Único	o atributo deve ser definido como único, quando seus valores não puderem ser repetidos.
Chave primária	atributo ou conjunto de atributos que definem um único registro (tupla) em uma entidade.
Chave estrangeira	atributo ou conjunto de atributos que garante o relacionamento entre duas ou mais entidades. Assegura o que chamamos de integridade referencial, isto é, se as tabelas devem ou não possuir uma informação cadastrada para permitir seu uso em outra tabela.
Valor default	indica se o atributo tem um valor padrão de preenchimento.
Regra de validação	demonstra se o atributo possui ou não uma regra de preenchimento. Por exemplo: o atributo sexo só pode receber valores M ou F.

Tabelas

São inúmeras as formas de apresentar as definições alinhadas, e utilizaremos a mais comum, a tabela. Então, por meio de tabelas, vamos criar o modelo lógico das entidades da solução imaginada para a padaria do senhor João. As entidades e relacionamentos com campos passam a ser chamados de tabelas e seus atributos de campos. Observe o exemplo, na tabela *Cartão*.

CARTAO

nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
codigo	INTEGER	8	Sim	Sim	Sim	Não	Não	Não
dt_Inicio_uso	DATE	8	Sim	Não	Não	Não	Não	Não
dt_fim_uso	DATE	8	Não	Não	Não	Não	Não	Não

Observação: O campo dt_fim_uso foi definido como não obrigatório, pois nenhum dos cartões, quando de seu cadastro, tem data indicando fim de prazo de validade. Veja as tabelas *Produto*, *Funcionario*, *Fornecedor*, *Fornece*, *ItemFornece* e *Compra*.

PRODUTO

nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
codigo	INTEGER	5	Sim	Sim	Sim	Não	Não	Não
nome	VARCHAR	40	Sim	Não	Não	Não	Não	Não
preco_venda	DECIMAL	10,2	Sim	Não	Não	Não	Não	Valor > 0
saldo	DECIMAL	10,2	Sim	Não	Não	Não	0	Valor >=0
estoque_minimo	DECIMAL	10,2	Sim	Não	Não	Não	0	Valor >=0

Observação: O campo saldo foi definido com decimal, porque nem sempre os produtos são vendidos em unidades.

FUNCIONARIO

nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
codigo	INTEGER	4	Sim	Sim	Sim	Não	Não	Não
nome	VARCHAR	50	Sim	Não	Não	Não	Não	Não
função	VARCHAR	30	Sim	Não	Não	Não	Não	Não

O melhor SGBD

Definir qual Sistema Gerenciador de Banco de Dados devemos adotar em uma solução informatizada nem sempre é fácil. Precisamos levar em conta seu preço, sua forma de comercialização, seus custos adicionais (valor da manutenção anual, preço por usuário etc.), estrutura de hardware que a solução demandará (rede, stand-alone, internet), grau de segurança, tipos de acesso, formas de gerenciamento de informações. Todos esses fatores têm de ser considerados, para chegar à melhor solução.

FORNECEDOR

nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
codigo	INTEGER	4	Sim	Sim	Sim	Não	Não	Não
nome	VARCHAR	50	Sim	Não	Não	Não	Não	Não
rua	VARCHAR	50	Sim	Não	Não	Não	Não	Não
complemento	VARCHAR	50	Não	Não	Não	Não	Não	Não
bairro	VARCHAR	40	Não	Não	Não	Não	Não	Não
cidade	VARCHAR	40	Sim	Não	Não	Não	Não	Não
estado	VARCHAR	2	Sim	Não	Não	Não	Não	Não
cep	VARCHAR	8	Sim	Não	Não	Não	Não	Não
telefone	VARCHAR	10	Não	Não	Não	Não	Não	Não
celular	VARCHAR	10	Não	Não	Não	Não	Não	Não

Observação: Um campo do tipo varchar permite até 255 caracteres, mas o tamanho de campos como rua, por exemplo, deve ir no máximo até 50 caracteres. Isso porque talvez seja preciso emitir etiquetas de endereçamento e como o faremos se o campo rua tiver 255 caracteres? O campo CEP também foi colocado como obrigatório, para que o usuário se lembre de preenchê-lo.

Os campos telefone e celular são do tipo varchar, pois o zero à esquerda é significativo, isto é, 01 é diferente de 1.

FORNECE

nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
numero_nota	INTEGER	7	Sim	Sim	Sim	Não	Não	Não
cod_fornecedor	INTEGER	4	Sim	Não	Sim	Sim	Não	Fornecedor já deve ser cadastrado
data	DATE	8	Sim	Não	Não	Não	Não	Não
valor_total	DECIMAL	10,2	Sim	Não	Não	Não	Não	Soma dos itens de itemfornece para esta nota.

ITEMFORNECE								
nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
numero_nota	INTEGER	7	Sim	Sim	Sim	Sim	Não	Não
cod_produto	INTEGER	5	Sim	Não	Sim	Sim	Não	Produto já deve ser cadastrado
quantidade	DECIMAL	5,2	Sim	Não	Não	Não	Não	Não
valor_unitario	DECIMAL	10,2	Sim	Não	Não	Não	Não	>0

COMPRA								
nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
numero	INTEGER	7	Sim	Sim	Sim	Não	Não	Auto-Numeração
cod_cartao	INTEGER	8	Sim	Não	Não	Sim	Não	Cartão já deve ser cadastrado
data	DATE	8	Sim	Não	Não	Não	Não	Não
forma_pagto	VARCHAR	10	Sim	Não	Não	Não	Não	Não
valor_total_compra	DECIMAL	10,2	Sim	Não	Não	Não	Não	Soma dos valores dos itens de itemcompra para esta compra
cod_func_caixa	INTEGER		Sim	Não	Não	Sim	Não	Funcionário deve ser cadastrado

ITEMCOMPRA								
nome	tipo de dados	tamanho	obrigatorio	unico	chave primaria	chave estrangeira	valor default	regra de validacao
numero_nota	INTEGER	7	Sim	Sim	Sim	Não	Não	Não
cod_produto	INTEGER	5	Sim	Não	Sim	Sim	Não	Produto já deve ser cadastrado
quantidade	DECIMAL	5,2	Sim	Não	Não	Não	Não	Não
valor_unitario	DECIMAL	10,2	Sim	Não	Não	Não	Não	Valor do campo preco_venda da tabela produto quando da inclusão da nota.

Observação: Um campo autonumeração é um campo numérico, que terá seu valor sequencial preenchido pelo próprio SGBD. Mas é importante frisar que nem todo Sistema Gerenciador de Banco de Dados (SGBD) possui esse campo. Por isso, é preciso verificar essa disponibilidade conforme o sistema adotado. Veja que o campo cod_func_caixa não estava definido no projeto lógico, mas, ao analisar as funcionalidades, verificou-se que o senhor João quer saber quem recebeu pela compra e, assim, foi necessário possibilitar o registro desta informação, a qual servirá também para indicar se determinada compra já foi paga ou não.

2.3.8. Projeto físico

O projeto físico consiste na tradução do modelo lógico para a linguagem SQL. Normalmente, é feito um script (lista dos comandos de criação do banco de dados e de suas tabelas dentro do SGBD). Os comandos para gerar as tabelas em SQL serão estudados no próximo capítulo, mas como exemplo, vem a seguir o projeto físico de nosso estudo de caso para o SGBD MySQL.

```
CREATE DATABASE Padaria;
USE Padaria;
```

```
CREATE TABLE Cartao (
codigo INTEGER NOT NULL PRIMARY KEY,
dt_inicio_uso DATE NOT NULL,
dt_fim_uso DATE);
```

```
CREATE TABLE Produto (
codigo INTEGER NOT NULL PRIMARY KEY,
nome VARCHAR(40) NOT NULL,
preco_venda DECIMAL(10,2) NOT NULL,
saldo DECIMAL(10,2) NOT NULL,
estoque_minimo DECIMAL(10,2) NOT NULL);
```

```
CREATE TABLE Funcionario (
codigo INTEGER NOT NULL PRIMARY KEY,
nome VARCHAR(50) NOT NULL,
funcao VARCHAR(30) NOT NULL);
```

```
CREATE TABLE Fornecedor (
codigo INTEGER NOT NULL PRIMARY KEY,
nome VARCHAR(50) NOT NULL,
rua VARCHAR(50) NOT NULL,
```

DICA

A definição das tabelas do modelo lógico deve ser feita com muita atenção e cuidado, pois podemos facilitar, e muito, a implementação da solução, por exemplo, incluindo campos como obrigatórios e definindo regras de inclusão e alteração. Dessa forma, colocamos no SGBD parte da responsabilidade pela consistência dos dados e aliviamos os programas que farão a entrada e a manipulação de dados.

```
complemento VARCHAR(50),
bairro VARCHAR(40),
cidade VARCHAR(40) NOT NULL,
estado VARCHAR(2) NOT NULL,
cep VARCHAR(8) NOT NULL,
telefone VARCHAR(10),
celular VARCHAR(10));
```

```
CREATE TABLE Fornece(
numero_Nota INTEGER NOT NULL PRIMARY KEY,
cod_Fornecedor INTEGER NOT NULL REFERENCES
Fornecedor(codigo),
data DATE NOT NULL,
valor_Total DECIMAL(10,2) NOT NULL),
PRIMARY KEY (numero_Nota,cod_Fornecedor));
```

```
CREATE TABLE ItemFornece (
numero_Nota INTEGER NOT NULL REFERENCES
Fornece(numero_Nota),
cod_Produto INTEGER NOT NULL REFERENCES
Fornecedor(codigo),
quantidade DECIMAL(5,2) NOT NULL,
valor_Unitario DECIMAL(10,2) NOT NULL,
PRIMARY KEY (numero_Nota,cod_Produto));
```

```
CREATE TABLE compra(
numero INTEGER NOT NULL PRIMARY KEY,
cod_Cartao INTEGER NOT NULL REFERENCES
Cartao(codigo),
data DATE NOT NULL,
forma_Pagto VARCHAR(10) NOT NULL,
valor_Total_Compra DECIMAL(10,2) NOT NULL,
cod_Func_Caixa INTEGER NOT NULL REFERENCES
Funcionario(codigo));
```

```
CREATE TABLE ItemCompra (
numero_Nota INTEGER NOT NULL REFERENCES
Compra(numero),
cod_Produto INTEGER NOT NULL REFERENCES
Fornecedor(codigo),
quantidade DECIMAL(5,2) NOT NULL,
valor_Unitario DECIMAL(10,2) NOT NULL,
PRIMARY KEY (numero_Nota,cod_Produto));
```


2.3.9. Análise funcional

Analisando os requisitos funcionais que encontramos na fase de análise de requisitos, serão escolhidas as rotinas a serem criadas para que todas as funcionalidades de nosso projeto sejam atendidas. Uma rotina do sistema pode automatizar mais de um requisito funcional anteriormente definido. Por exemplo, a digitação de uma nota fiscal de compra de mercadorias para a padaria do senhor João implementará não apenas as tabelas Fornece e ItemFornece, mas atualizará o saldo do produto na tabela Produtos, podendo alterar também o valor do campo preço de venda daquele item. Se analisarmos os requisitos funcionais do sistema proposto para a padaria teremos, pelo menos, as seguintes rotinas:

- **Manutenção de cartões**
- **Manutenção de funcionários**
- **Manutenção de fornecedores**
- **Manutenção de produtos**
- **Registro de produto vendido pelos atendentes**
- **Apresentação da somatória da compra, recebimento da compra e marcação de qual funcionário recebeu determinada compra**
- **Lançamento da Nota Fiscal de Entrada**
- **Controle de acesso do usuário**
- **Opções de seleção de rotinas disponíveis para o usuário do sistema, de acordo com seu nível de acesso**
- **Consulta de preços de produtos.**

Após a definição das rotinas e dos requisitos funcionais que elas implementarão, deve-se separá-las em módulos de acordo com a característica de cada uma. Assim, formam-se grupos de rotinas que implementam requisitos semelhantes, a fim de se criar uma estrutura lógica e funcional de acesso às principais funcionalidades do sistema, permitindo, também, o controle de acesso a tais funcionalidades.

No caso do nosso exemplo fictício, a padaria do senhor João, temos alguns grupos de requisitos que tratam da manutenção das tabelas cadastrais do sistema – as tabelas de produto, funcionário, cartão e fornecedor. Podemos nominar as funcionalidades comuns a essas rotinas como, por exemplo, a manutenção de informações cadastrais. Pode-se dizer que os requisitos de registrar um produto em um cartão, lançar uma nota fiscal de um fornecedor, ou registrar o pagamento de uma venda descrevem a movimentação da padaria, logo cabe incluí-los no grupo de rotinas de movimentação do sistema da padaria. Haverá, ainda, grupos para as consultas e relatórios do sistema, nos quais serão reunidas as rotinas que implementarão essas funcionalidades.

A engenharia de software oferece várias técnicas para separar as rotinas do sistema em módulos, como o Diagrama Hierárquico de Funções ou os Diagramas de Pacotes e de Componentes, que veremos no capítulo 4.

2.3.10. Projeto de programas da aplicação

A partir de agora, vamos utilizar vários conceitos relacionados ao desenvolvimento de software. Primeiro será preciso definir a linguagem de programação que adotaremos para implementar a solução. Para isto, temos de levar em conta o ambiente em que o sistema será implementado, ou seja, o sistema operacional

instalado. Também é preciso considerar a estrutura de hardware definida para a solução: se o programa será implantado em uma rede de computadores, qual é sua arquitetura e onde ficarão instalados a aplicação e o Sistema Gerenciador de Banco de Dados (SGDB), tema, aliás, que veremos mais adiante.

Definir um padrão para a interface com o usuário (isto é, das telas a serem exibidas pelo sistema), assim como fontes, cores, formato e tamanho dos elementos das telas (como botões, barras de menu, caixas de texto etc.), requer o conhecimento prévio de detalhes da estrutura em que nossa solução será implementada. Nem sempre os diversos ambientes nos permitem trabalhar com todos esses recursos.

É interessante definir essa interface com o usuário e para isso geralmente usamos a técnica de prototipagem, definida no capítulo 1. Assim, montamos um pequeno protótipo só com a tela a ser exibida e o apresentamos ao cliente para que ele tenha uma ideia de como a visualizará e de como poderá interagir com as informações que solicitou.

É o momento, então, de definir o funcionamento de cada programa a ser criado para a implementação da solução. Também para essa tarefa existem várias técnicas, tais como a descrição em português estruturado do funcionamento de cada programa, o uso de diagramas da UML (diagrama de classes, de sequência, e de máquina de estados, que serão vistos no capítulo 4 deste livro). O que deve ficar claro é como o programa será construído para implementar a rotina definida, em relação à interface com o usuário e ao seu próprio funcionamento, a conexão com o sistema gerenciador de banco de dados e a arquitetura de hardware e software onde a solução será instalada.

Devemos definir também o plano de testes a que devem ser submetidos cada um dos programas em vias de criação, a fim de validar seu correto funcionamento.

2.3.11. Implementação da Transação

Com as definições em mãos, partimos para a programação e os testes das rotinas na linguagem definida. À medida em que essas rotinas vão sendo concluídas, devemos elaborar um ambiente de testes que simule o meio do usuário para que se possa analisar o funcionamento do sistema como um todo, isto é, com todos os programas em funcionamento. Nesse momento, é importante certificar-se de que todos os programas estão executando suas tarefas corretamente, de que o banco de dados está sendo atualizado de forma consistente e de que a estrutura de hardware e software está permitindo o funcionamento correto do sistema e oferecendo ao usuário as informações que ele solicitou, no tempo que precisa delas.

Na fase seguinte, devemos iniciar o treinamento dos usuários, para que possam operar o sistema. No caso da padaria do senhor João, cada atendente tem de saber como se identificar no sistema e como registrar as compras dos clientes no balcão. Os caixas têm de aprender como se registra o recebimento de uma compra e como se lança notas fiscais de fornecedores. E, claro, seria preciso treinar o senhor João, quanto a todas as tarefas do sistema. Tanto no exemplo fictício como em um projeto real, quando todas as informações estiverem cadastradas e os cadastros de funcionários, produtos, cartões e fornecedores estiverem corretos, é hora de estudar a melhor data para a implantação, caso ainda não tenha sido definida. É importante, ainda, estabelecer uma rotina de cópia das informações do sistema (backup). O usuário responsável por essa tarefa tem de ser treinado e instruído sobre sua importância, pois com esse recurso se reduzem as perdas de informação, no caso de falhas de hardware ou software. Na data da implantação e nos dias subsequentes é fundamental acompanhar o funcionamento do sistema, efetuando pequenos ajustes, caso seja preciso. Terminada essa fase, podemos concluir que o sistema está entregue.

Informática, psicologia, administração

Você aprendeu a usar a metodologia para projetar, construir e implementar um sistema. Tarefa cumprida? Ainda não. O trabalho só estará completo quando, além da informática, você lançar mão de seus conhecimentos em psicologia e administração, para lidar com as pessoas envolvidas no projeto e suas expectativas – o que, convenhamos, não é algo simples. Ao projetar e construir um sistema, é preciso ter sempre em mente que os usuários estão ansiosos por vê-lo funcionando em seu ambiente. É possível usar diversas ferramentas para que sua execução seja transparente, o que ajuda a amenizar as expectativas. Pode-se, por exemplo, adotar um cronograma que lhes permita acompanhar todas as fases do projeto, de modo que percebam claramente quais passos já foram dados e os que virão na sequência. Esse, porém, é apenas um dos exemplos de ferramentas auxiliares. O importante é manter a disposição de pesquisar outras, para que o desenvolvimento de nossos projetos se torne cada vez mais rápido e claro.

TETRA IMAGES/CORBIS/LATINSTOCK

Figura 36 Execução de um projeto.

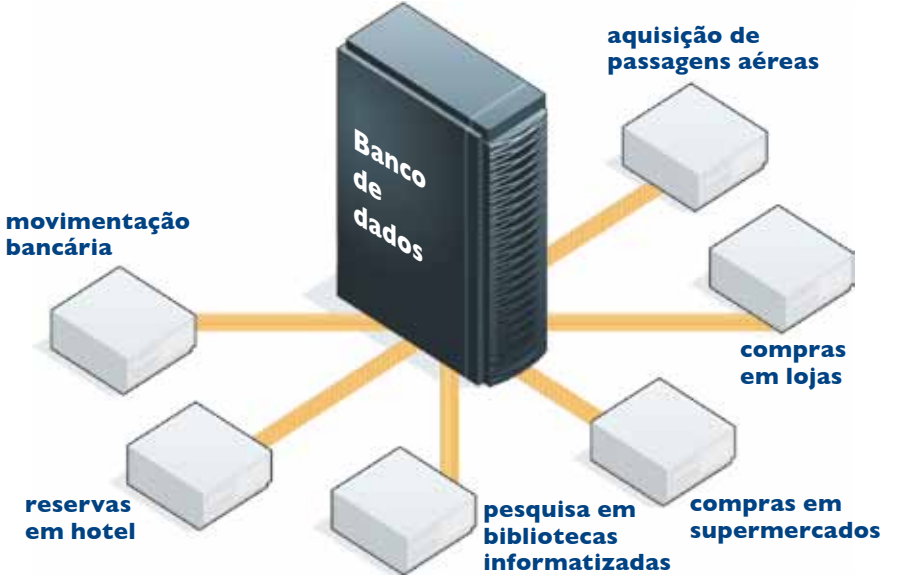
Capítulo 3

Banco de dados

- Evolução dos sistemas de computação
- Conceitos e terminologia
- Abordagem relacional
- Administração e gerenciamento

Segundo Ramez Elmasri, autor de vários livros sobre informática, o primeiro Sistema Gerenciador de Banco de Dados (SGBD) surgiu no final de 1960, com base nos primitivos sistemas de arquivos disponíveis na época, incapazes de controlar o acesso concorrente por vários usuários ou processos. Assim, os SGBDs evoluíram de sistemas de arquivos para armazenamento em disco, quando foram criadas novas estruturas de dados com o objetivo de armazenar informações. Os SGBDs evoluíram ao longo do tempo, com a introdução de diferentes formas de representação, ou modelos de dados, para descrever a estrutura das informações neles contidas.

Os bancos de dados se tornaram um componente essencial e indispensável em nosso dia a dia. Geralmente não nos damos conta disso, mas sem eles não conseguiríamos mais fazer atividades corriqueiras como depósitos, saques ou quaisquer outras formas de movimentação bancária, reservas em hotel, pesquisa em bibliotecas informatizadas, compras em supermercados ou lojas, aquisição de passagens aéreas. Tais atividades são exemplos clássicos de aplicações que utilizam bancos de dados para manipular textos, valores, imagens etc (figura 37).

Figura 37
Banco de dados.

3.1. Evolução dos sistemas de computação

Os últimos anos têm sido pródigos, em termos da tremenda evolução observada nos sistemas computacionais. O avanço nas diversas versões dos sistemas operacionais permite que haja cada vez mais e mais recursos disponíveis e novas metodologias de acesso, que ampliam a velocidade e a forma de usá-los. Entre as novidades estão os bancos de dados pós-relacionais, também chamados de bancos de dados orientados a objetos. Tudo isso tem um grande impacto, também, na construção de sistemas de aplicação. Vale, portanto, analisar as metodologias empregadas na implantação de sistemas comerciais baseados em computadores.

3.1.1. Abordagem tradicional

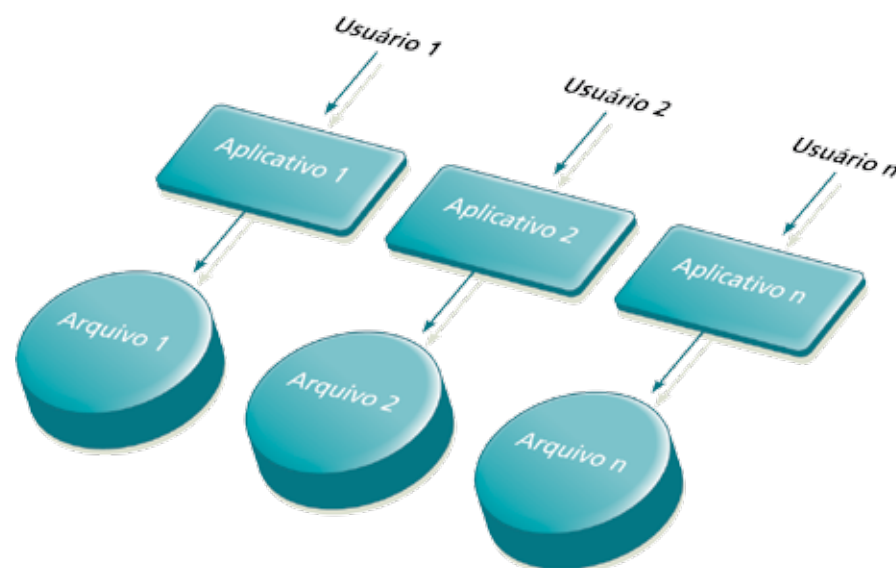
Segundo o professor e pesquisador Abraham Silberschatz (1999), a **abordagem tradicional** é baseada em técnicas não estruturadas, utilizando a intuição natural do analista de sistemas, bem como em sua capacidade criativa e vivência no ambiente do sistema, ou seja, em seu conhecimento de negócios. A principal característica dessa abordagem é a orientação intuitiva desses profissionais para desenhar o sistema, com base em procedimentos executados e descritos pelo usuário em sua forma pura, sem orientação a dados e otimizações organizacionais. Nesse período utiliza-se uma mistura de linguagens da terceira geração e de linguagens procedurais, com necessidade de declaração de dados e identificação física da localização da informação. Essa abordagem não prioriza a integração da informação, mas sim a solução de problemas setorizados, ou seja, pode gerar retrabalho, pois muitas vezes haverá os mesmos dados em diversos sistemas na organização.

As principais características da abordagem tradicional são: cada aplicação tem seus próprios arquivos; os dados são repetitivos; inconsistência; subordinação de programas a arquivos; manutenção difícil e cara; o analista é o "dono" do sistema; falta de integridade e de segurança.

Assim, sob a abordagem tradicional desenvolvem-se sistemas e aplicações com arquivos de propriedade da aplicação e de seus programas, em função da utilização de linguagens de terceira geração, sem se levar em consideração a repetição dos dados em outros sistemas.

Como consequência, temos a redundância de informações, a qual provoca total incoerência e muitas vezes inconsistência de dados. Os sistemas são elaborados pensando-se na distribuição física dos arquivos (por exemplo, em que pasta ou diretório esse arquivo está fisicamente no HD), criando-se, assim, uma dependência dos arquivos pelos sistemas. Segundo Elmasri (1999), isso significa que qualquer alteração na base de dados implica a adaptação de grande quantidade de programas e, portanto, significa enorme esforço de programação. É bom frisar que, sempre que houver grandes alterações, temos uma forte tendência de esquecer alguns procedimentos, ao realizar alguma modificação, o que pode resultar em erros que muitas vezes demoramos para perceber, pois o sistema poderá continuar gravando as informações na base de dados anterior.

Caso haja muitos programas desenvolvidos dessa maneira, o sistema de tecnologia da informação da organização é ineficiente, pois se trata de uma abordagem antiga e inadequada ao atual estágio de desenvolvimento de sistemas. Esse tipo de abordagem pode ser visualizada da seguinte forma: cada usuário utiliza um aplicativo (ou um conjunto de aplicativos), o qual, por sua vez, usa um arquivo de dados que muitas vezes não é acessado por outro usuário, como sugere a ilustração a seguir (figura 38).

Figura 38
A abordagem tradicional: para cada usuário, um ou mais aplicativos.

3.1.2. Abordagem de sistemas integrados

Elmasri (1999) diz ainda que, por causa das deficiências da abordagem anterior, criou-se, por volta de 1980, uma nova, a de **sistemas integrados**, a qual pretendia resolver as questões de redundância e integridade de informações, trazendo uma visão corporativa de sistemas. Ela se vale da percepção de que existem conjuntos de arquivos de dados de interesse comum às várias áreas de uma organização, ou seja, de que não precisamos redigitar informações que já foram digitadas por outros usuários de outros departamentos ou usuários de outros sistemas. A integração entre os sistemas mantém um grau acentuado de unicidade da informação dentro dos sistemas da organização. Assim, tal abordagem eliminou a redundância de dados que caracteriza a anterior. Aqui, os sistemas passam a utilizar uma única base de informações, sem a repetição de arquivos por sistema.

As principais características da abordagem de sistemas integrados são: pouco adaptável; as alterações comprometem vários sistemas; aplicações estáticas com o tempo; problemas em um sistema paralisam atividades de outros; programas ainda subordinados a arquivos; visão de usuário inexistente; integridade e segurança são fracas.

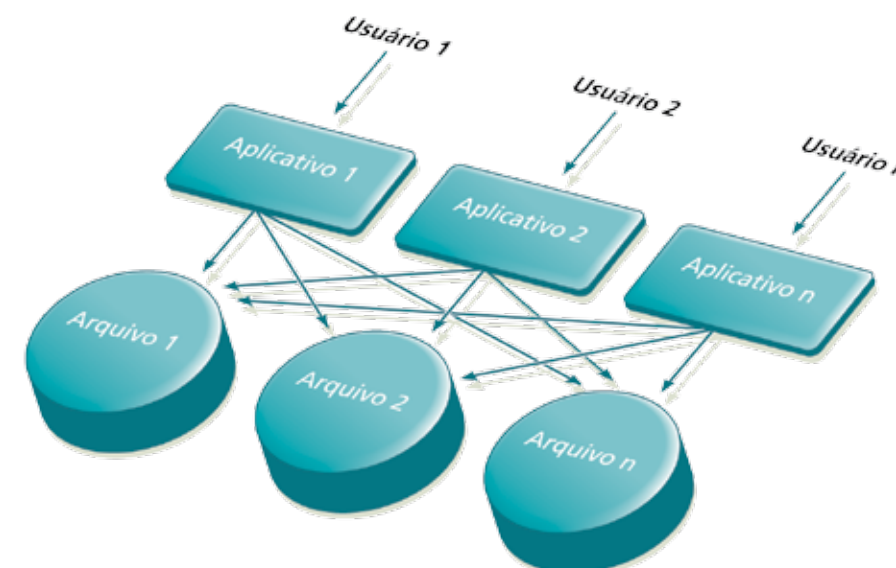
Os sistemas desenvolvidos dessa maneira iniciaram a integração no universo corporativo, ficando dependentes uns dos outros por utilizarem a mesma base de dados. Assim, alterações em qualquer arquivo de dados implicavam modificações em programas de mais de um dos sistemas que acessavam a mesma base, em virtude de utilizar linguagens de alta dependência de dados (figura 39). Com o crescimento das aplicações, essa abordagem também se tornou ineficiente. Qualquer alteração na base de dados afetava um número ainda maior de sistemas e programas. Com tantas dificuldades relacionadas ao volume de alterações, as aplicações acabaram por se tornar praticamente estáticas, levando os departamentos de tecnologia da informação à estagnação e a serem considerados ineficientes pelos gestores das organizações.

3.1.3. Abordagem de bancos de dados

Segundo Elmasri (1999), embora haja diferenças até certo ponto significativas entre as duas abordagens apresentadas anteriormente, ambas têm em comum uma característica determinante: a ênfase dada aos processos, ao desenvolver uma aplicação. É por isso que surgem as dificuldades de manutenção, ocasionadas parcialmente pelo forte acoplamento dos programas aos arquivos. A abordagem de banco de dados, ainda considerada a mais adequada, surgiu da necessidade de desenvolvimento de sistemas cada vez mais complexos e flexíveis, integrando os dados de toda a organização e não mais apenas os sistemas departamentais. Sob esse prisma, a ênfase no desenvolvimento de uma aplicação não é mais sobre os processos, e sim sobre os dados. Podemos dizer que, para a abordagem tradicional e de sistemas integrados, na expressão "processamento de dados" a parte mais importante é o processamento, enquanto na abordagem de banco de dados a palavra fundamental é dados.

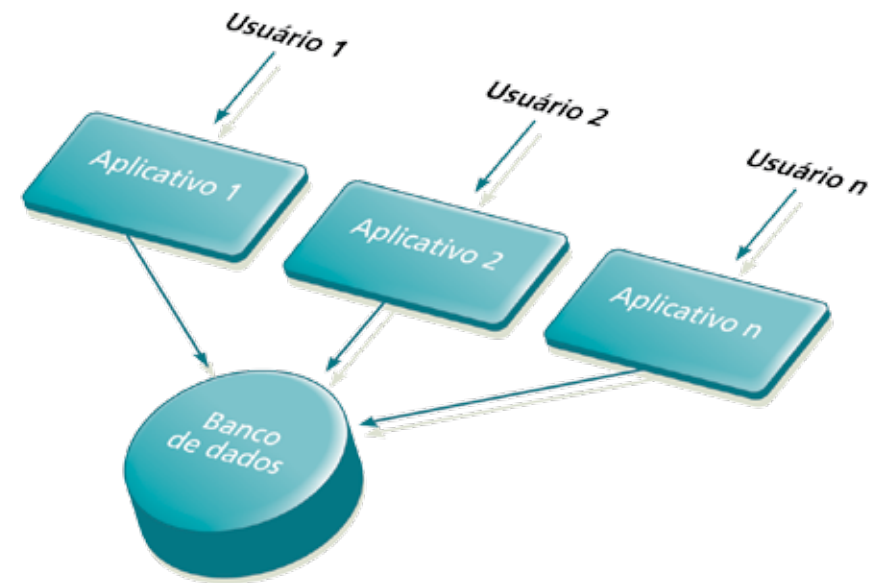
Segundo Elmasri (2002), há duas preocupações principais, quando trabalhar com essa abordagem. A primeira, em um nível mais conceitual, consiste na definição de um modelo de dados em termos da organização como um todo, abordando seus diversos setores. É preciso retratar todo o interrelacionamento entre as diversas áreas e departamentos que compõem o negócio, bem como todas as necessidades de informação de cada um desses setores. Somente com base nisso poderemos garantir que as aplicações desenvolvidas individualmente terão o comportamento desejado quanto à integração com outros sistemas e, também, que os dados serão os mais confiáveis.

A segunda preocupação diz respeito à forma de implementação física dos dados, em relação aos programas que irão manipular. Aqui deveremos levar em consideração as linguagens de programação disponíveis e qual delas será adotada, tendo sempre o cuidado de selecionar a melhor para o tipo de sistema computacional que estamos desenvolvendo. É necessário que os programas tenham uma visão de alto nível dos dados, o mais independente possível dos fatores físicos ligados à forma de armazenamento desses dados. Assim, nos Sistemas Gerenciadores de Banco de Dados (SGDB), há uma separação entre o local onde se armazena o banco de dados, bem como a definição física dos dados, e aquele no

Figura 39
Abordagem de sistemas integrados: dependência.

qual eles são realmente utilizados. Daí se originam, inclusive, os conceitos de Linguagem de Definição dos Dados e de Linguagem de Manipulação dos Dados. Passa a existir, assim, um repositório onde são armazenados esses elementos de mais baixo nível, de forma que fiquem externos ao código do sistema. Esse repositório apresenta-se na forma de um dicionário de dados com aparência e transparência irrelevantes para o analista de sistemas e os programadores.

VANTAGENS	REQUISITOS
• Desenvolvimento flexível e produtivo	• Aquisição e utilização de um SGBD
• Integração dos dados em nível empresarial	• Utilização de dicionário de dados
• Transparência dos dados às aplicações	• Centralização da definição de dados
• Maior controle sobre a integridade dos dados	• Centralização do projeto das bases de dados
• Maiores controles de segurança dos dados	

Também devemos avaliar com muita atenção qual arquitetura de hardware e software será a mais adequada ao nosso projeto. A abordagem de banco de dados está ilustrada na figura 40. Trabalhar com ela, no desenvolvimento de sistemas, exige dos profissionais de tecnologia da informação comprometimento com alguns princípios.

Figura 40
Abordagem de banco de dados.

3.2. Conceitos e terminologia

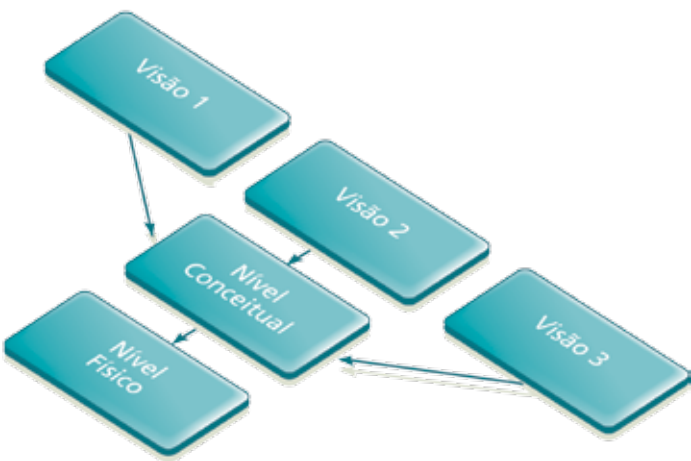
Para trabalhar essa abordagem é preciso conhecer os conceitos e as terminologias relacionadas a banco de dados, justamente os temas abordados neste capítulo.

3.2.1. Abstração de dados

De acordo com Elmasri (2002), a abstração de dados é a capacidade de modelar as características, variáveis e constantes, de forma a desprezar os dados que não interessam (veja quadro *Os níveis de abstração*). Um SGBD é composto por uma coleção de arquivos interrelacionados e de um conjunto de programas que permitem aos usuários acessar e modificar os arquivos. Sua proposta maior é prover os usuários de uma visão abstrata dos dados. Ou seja, o sistema omite alguns detalhes de como as informações são armazenadas e mantidas, mas, para que possa ser utilizado em projetos bastante complexos, esses dados devem ser recuperados de maneira eficiente. Como nem sempre os usuários de bancos de dados são treinados em informática, a complexidade fica encapsulada em diversos níveis de abstração, para simplificar a interação desses usuários com o sistema (DATE, 2000). Há três níveis de abstração e seu interrelacionamento é ilustrado na figura 41.

OS NÍVEIS DE ABSTRAÇÃO

- **Nível físico:** é o mais baixo, no qual se descreve como os dados são armazenados, onde essas complexas estruturas são descritas detalhadamente. Assim, os registros – de um cliente, de um fornecedor, de uma conta bancária – são descritos como um bloco de posições consecutivas de armazenamento, como por exemplo string ou byte.
- **Nível conceitual:** é o próximo nível de abstração, em que se descreve quais dados são armazenados e as relações existentes entre eles. Aqui, o banco de dados é descrito em termos de um pequeno número de estruturas simples. É utilizado pelos administradores de banco de dados, que podem decidir quais informações devem ser mantidas. No nível conceitual, os registros – de um cliente, de um fornecedor, de uma conta bancária – se interrelacionam.
- **Nível visual:** é o mais alto de abstração e descreve apenas a parte do banco de dados. Muitos usuários não estão interessados em todas as informações existentes. Sua definição serve para simplificar a interação deles com o sistema, que pode fornecer várias visões diferentes para o mesmo banco de dados. Por exemplo: cada categoria de funcionário pode visualizar determinado grupo de informações.

Figura 41
Interrelacionamento entre os níveis de abstração.

Henry Korth é autor de vários livros clássicos da informática.

3.2.2. Instâncias e esquemas

Os bancos de dados se alteram, ao longo do tempo, à medida que informações são inseridas ou deletadas. O conjunto de dados armazenados em determinado momento é conhecido como instância do banco de dados. Já o projeto do banco de dados é chamado de esquema de banco de dados (DATE, 2000).

Segundo **Korth** (1995) o conceito de um esquema de banco de dados corresponde à noção de definição de tipo na linguagem de programação. Uma variável de um dado tipo tem um valor particular em um determinado instante de tempo. Assim, o conceito de valor de uma variável na linguagem de programação corresponde ao conceito de uma instância de um esquema de banco de dados. Ainda de acordo com Korth (1995), os sistemas de bancos de dados possuem diversos esquemas, subdivididos de acordo com os níveis de abstração descritos anteriormente. No nível mais baixo está o esquema físico, no intermediário o esquema conceitual e no mais alto, um subesquema. Em geral, os sistemas de bancos de dados suportam um esquema físico, um esquema conceitual e diversos subesquemas.

- **Independência física: é a habilidade de modificar o esquema físico sem que seja preciso modificar os programas. Alterações no nível físico podem ser necessárias somente para uma ocasional melhoria de performance.**
- **Independência lógica: é a possibilidade de modificar o esquema conceitual sem que isso demande modificação nos programas. É necessária apenas quando a estrutura lógica do banco de dados é alterada. Por exemplo, se forem criadas mais colunas em uma determinada tabela do banco de dados. A independência lógica é mais difícil de alcançar do que a independência física, porém os programas são bastante dependentes da estrutura lógica dos dados que acessam**

3.2.3. Independência de dados

Independência de dados é a habilidade de modificar a definição de um esquema em um nível, sem afetar sua definição de esquema em um outro nível, mais alto. Essa independência constitui o principal objetivo a ser alcançado no desenvolvimento de um banco de dados, pois permitirá a expansão das atividades da empresa, facilitando o desenvolvimento de novos sistemas. Tal independência consiste na capacidade de favorecer que haja evolução na descrição de dados, como a criação de uma nova estrutura lógica decorrente de uma nova aplicação, ou inclusão de um dado novo numa estrutura existente, sem que os sistemas ou aplicações (em última análise, os programas) tenham de ser alterados. Essa capacidade deve se estender aos casos de mudança na organização dos arquivos, em consequência de degradação e perda de eficiência (DATE, 2000). De acordo com esse conceito, os programas interagem com a visão lógica do banco de dados. A localização física do dado é totalmente transparente.

3.2.4. Linguagem de definição de dados

Um esquema de banco de dados é especificado por um conjunto de definições, por meio da chamada linguagem de definição de dados, ou DDL (do inglês, Data Definition Language). A compilação de comandos DDL é um conjunto de tabelas armazenadas em um arquivo chamado dicionário (ou diretório) de dados. Trata-se de um arquivo que contém metadados (dados sobre dados) e é sempre consultado antes que haja qualquer leitura ou modificação pelo banco de dados. Os comandos DDL servem para definir esquemas, remover relações, criar índices e modificar esquemas de relação.

3.2.5. Linguagem de manipulação de dados

A DML (do inglês Data Manipulation Language) decorre do fato de os níveis de abstração não se aplicarem somente à definição ou à estruturação de dados, mas também à sua manipulação. Isso tem uma série de significados, no âmbito do banco de dados: como **recuperamos a informação armazenada**, como inserimos novas informações, como deletamos as informações, como modificar dados armazenados.

Uma consulta (também chamada de query) é um comando que requisita o resgate de uma informação, com base na álgebra relacional e no cálculo relacional de tupla.

3.2.6. Usuários de banco de dados

Já sabemos que o objetivo central de um banco de dados é prover uma solução para armazenamento e busca de informações por seus usuários. Estes, os que irão interagir com o banco de dados para realizar essas operações, podem ser classificados em cinco tipos, de acordo com o tipo de interação (DALTON, 2008). Confira quais são:

- **Profissionais de sistemas informatizados:** aqueles que desenvolvem aplicativos desenvolvidos para realizar a interação com os bancos de dados.
- **Programadores ou desenvolvedores:** são os que fazem a interação com os sistemas, utilizando-se das DML, que são incluídas nos programas, denominados programas de aplicação.
- **Usuários avançados:** interagem com o sistema sem escrever programas. Como têm conhecimentos de linguagem de consulta, realizam chamadas específicas nessa linguagem para atender a suas demandas.
- **Usuários leigos:** só interagem com os sistemas desenvolvidos pelos programadores.
- **Administradores de banco de dados (DBA, do inglês Data Base Administrator):** são os responsáveis pela criação, manutenção e bom funcionamento do banco de dados. Avaliam sua performance com frequência e, sempre que necessário, tratam de aprimorá-la.

3.3. Abordagem relacional

Em 1969, o pesquisador da IBM Edgar Frank Codd, já mencionado no capítulo 2, começou a pesquisar uma maneira melhor de organizar dados e introduziu o conceito de banco de dados relacional, que é organizado em tabelas simples. Conforme Elmasri (2002), Codd criou um modelo que permite aos projetistas dividir suas bases de dados em tabelas separadas, mas relacionadas, de modo

a otimizar a eficiência da execução, mantendo a mesma aparência externa do banco de dados original para os usuários. Por isso é considerado o pai do banco de dados relacional. A maioria dos bancos de dados é relacional. Baseia-se no princípio de que as informações em uma base de dados podem ser consideradas como relações matemáticas e são representadas de maneira uniforme. Os objetos manipulados pelos usuários são as linhas (tuplas, na terminologia original) e as colunas das tabelas. Para fazer isso o usuário conta com um conjunto de operadores e funções de alto nível, denominados álgebra relacional.

3.3.1. Características principais

As principais características da abordagem relacional baseiam-se em uma série de fatores (KORTH, 1995), definidos a seguir.

- **Estrutura de dados tabular:** os dados são representados em forma de tabela, que é denominada relação, constituída de linhas ou tuplas, colunas ou atributos e, finalmente, tipos de dados que podem aparecer em uma coluna específica, chamada domínios. Assim, não existe o conceito de item de grupo, em um banco de dados relacional. Todos os itens de uma tabela são elementares.

- **Álgebra relacional:** a manipulação das tabelas é realizada por operadores por meio dos quais podemos acessar dados de diversas maneiras, ou seja, é um conjunto de operações e relações. Cada operação usa uma ou mais relações como seus operandos e produz outra relação como resultado. Dessa forma, a recuperação das informações em um SGBD relacional se faz em conjuntos de registros que comporão uma nova relação.

- **Dicionário de dados ativo e integrado:** um banco de dados que mantém as descrições dos objetos existentes no sistema da empresa deve ser ativo, no sentido de estar sempre disponível para utilização, e integrado, na medida em que englobe todas as informações sobre o sistema, permitindo realizar referências cruzadas nos dados.

- **Consistência automática de campos:** o próprio sistema tem a funcionalidade de validar os dados nos quesitos de tamanho, formato, tipo e integridade.

- **Referência cruzada de dados:** permite relacionar linhas e colunas, ou seja, valores de campos com linhas;

- **Integridade dos dados:** para garantir a qualidade das informações do banco de dados, devemos nos preocupar com quatro categorias de integridade dos dados:
  1. **De entidade:** definimos uma linha como exclusiva de determinada tabela, ou seja, definimos uma **chave primária (Primary Key)** de uma tabela.
  2. **Integridade de domínio:** valida entradas de uma coluna específica, podendo restringir suas relações em linhas, quando estas são incluídas ou excluídas. São as chaves estrangeiras (**Foreign Key**).
  3. **Integridade referencial:** preserva as relações definidas entre tabelas, permitindo consistência em todas elas.
  4. **Integridade definida pelo usuário:** o usuário define suas regras de negócio, quando não se enquadra nas outras categorias de integridade apresentadas.

A chave primária (Primary Key) serve para identificar cada registro numa tabela. Pode ser um atributo ou uma combinação de atributos.

A chave estrangeira (Foreign Key) pode ser uma coluna ou combinação de colunas usada para estabelecer links entre duas tabelas.

3.3.2. Princípios da orientação

O conceito de visões de dados tem a ver com a forma como o usuário os vê. Existe a possibilidade de criar visões de subconjuntos dos dados colocados em um SGBD. Os usuários podem ter acesso a parte do modelo, independentemente da forma como estão contidos no banco de dados. Tomemos como exemplo um banco de dados com clientes e pedidos. Podemos montar uma visão de informações que mostre os clientes com seus respectivos pedidos, sem que isso afete as estruturas do modelo de dados implementado ou apresentar uma visão com informações cadastrais de clientes sem dados financeiros, por exemplo. As visões são formadas conforme a necessidade do usuário. Para definir essas visões é preciso atentar para dois fatores: a transparência de dados (como e onde estão os dados torna-se secundário para o usuário, irrelevante) e os elos implícitos (colunas comuns nas tabelas, sem restrição a tipo de relacionamento).

3.3.3. As doze regras de Edgar F. Codd

As bases da abordagem relacional, como sabemos, foram lançadas em 1970 por Edgar F. Codd. Na época o pesquisador estabeleceu um conjunto de doze regras para um banco de dados realmente relacional. Segundo Date (2000), as normas de Codd, que vamos conhecer a seguir, discutem a fidelidade de um SGBD ao modelo relacional.

Regra número 1 - Representação de valores em tabelas

Toda informação, em um Banco de Dados Relacional, é apresentada em nível lógico por valores em tabelas.
Os nomes das tabelas, colunas e domínios são representados por séries de caracteres que, por sua vez, devem também ser guardadas em tabelas, as quais

BENEFÍCIOS DA ABORDAGEM

- Independência dos dados
- Visões múltiplas de dados
- Melhor comunicação entre o departamento de informática e os usuários
- Redução acentuada do desenvolvimento de aplicações e do tempo gasto em manutenção de sistemas
- Melhoria na segurança de dados
- Mais agilidade gerencial de informações ligadas ao processo decisório da empresa

formam o catálogo do sistema, o dicionário de dados. Portanto, o princípio de que a informação deve estar sob a forma de tabela é extensiva ao dicionário de dados. As tabelas são bidimensionais, o que equivale a dizer que não há colunas repetitivas (Cláusula Occurs, por exemplo, não existe).

Regra número 2 - Acesso garantido

Todo e cada dado num Banco de Dados Relacional tem a garantia de ser logicamente acessível, recorrendo-se a uma combinação do nome da tabela, um valor de chave primária e o nome da coluna.
Isso significa que a ordem das linhas, assim como a das colunas, é irrelevante. Para obter uma informação específica, ou seja, o valor de uma coluna em uma linha de uma tabela, basta informar o nome da tabela, o valor de uma chave primária (no caso, da linha a ser recuperada) e o nome da coluna da tabela em questão. Da mesma forma, para saber quais ocorrências de uma tabela têm um determinado valor em uma coluna, basta informar o nome da tabela e o nome da coluna, que o conteúdo poderá ser comparado. Conclusão: a regra especifica que, em um banco de dados efetivamente relacional, não existe informação inacessível.

Regra número 3 - Tratamento sistemático de valores nulos

Valores nulos são suportados em um SGBD relacional nulo para representar exatamente a informação perdida ou inexistente, inaplicável.
Para mais bem entender essa regra, vale a pena definir o que é valor nulo: é aquele utilizado para representar uma informação perdida ou desconhecida. Conceitualmente, um valor é inexistente. A inexistência de conteúdo em um campo equivale a dizer que seu valor é nulo. Mas atenção: jamais confunda zero(s) ou espaço(s) em branco com strings vazios.

Imagine um pacote de supermercado de papel, desdobrado. Sem tocá-lo ou olhar em seu interior, você não pode saber se está vazio ou contém algum produto. Portanto, nesse instante, para você, o conteúdo do pacote é NULO. É uma informação desconhecida, inexistente. Valores nulos, portanto, são suportados por um SGBD para representar exatamente a informação perdida ou inexistente, inaplicável. Para suportar a integridade de identidade é preciso especificar "não são permitidos valores nulos" em cada coluna da chave primária e em qualquer outra coluna que se considerar como uma restrição de integridade. Se levarmos em conta que um atributo tem de, obrigatoriamente, possuir conteúdo, estaremos especificando que seu valor não pode ser nulo. Assim, o SGBD precisa reconhecer a informação nula para poder restringi-la ou aceitá-la, se necessário.

Regra número 4 - Catálogo relacional ativo baseado no modelo relacional

A descrição do banco de dados é representada em nível lógico, da mesma maneira que seus dados.
Esta regra exige que um SGBD relacional tenha uma mesma linguagem para acesso e definição dos dados no dicionário e para a manipulação dos dados do banco e do dicionário. Por exemplo: um comando para adicionar informações em uma tabela da aplicação deve ser o mesmo para acrescentar informações no dicionário de dados.

Regra número 5 - Sublinguagem detalhada e dados

Um sistema de banco de dados relacional deve ter uma linguagem cujas instruções, com sintaxe bem definida, suportem a definição de dados, a definição de visão de dados, a manipulação de dados, as restrições de integridade, as autorizações e os limites de transação.

Instruções

- **Definição dos dados:** criar ou adicionar tabelas no dicionário de dados.
- **Definição de visão de dados:** fazer operações relacionais de junção, criar visões de partes do modelo de dados.
- **Manipulação de dados:** permitir criar, alterar, consultar e deletar conjuntos de dados.
- **Restrições de integridade:** possibilitar que sua sublinguagem controle as restrições específicas de uma tabela ou de valores aceitáveis para uma determinada coluna.
- **Autorizações:** definir limites de acesso a tabelas por usuário, inclusive em nível de coluna.
- **Limites de transação:** tornar viável a delimitação de uma transação lógica de modificação do banco de dados, o que fornece um alto nível de segurança da integridade de informações no banco de dados.

Regra número 6 - Atualização de visão

Todas as visões de dados, que são teoricamente atualizáveis, são também atualizáveis pelo sistema.
Atualizar significa mais do que simplesmente modificar a informação; abrange também a inclusão e/ou exclusão de dados. Por exemplo, se definimos uma visão de dados como um subconjunto horizontal de uma tabela, deve ser possível adicionar dados a essa visão. Consideremos uma tabela de funcionários e que haja visões de funcionários técnicos, funcionárias secretárias e funcionários administrativos. Atualizar uma dessas visões quer dizer, entre outras possibilidades, adicionar dados para a inclusão de uma secretária ou de um técnico. Outro exemplo seria criar uma visão de uma tabela de médicos por especialidade, digamos cem médicos em uma tabela, dos quais trinta são especializados em pediatria. Ao criarmos a visão Pediatras, uma seleção de linhas, que conterá somente os médicos com esta especialidade, cria-se no dicionário de dados um sinônimo de médico, denominado Pediatra, que tem como condição para esse role (literalmente, papel) o valor da coluna especialidade ser igual a pediatria. Esta visão deve ter a possibilidade de ser deletada. Se comandarmos DELETAR PEDIATRIA ou ALTERAR um pediatra específico, tais atualizações devem ser realizadas diretamente na tabela originária da visão.

Regra número 7 - Atualização de alto nível

No banco relacional, a linguagem de alto nível se aplica não somente à

consulta de dados, mas também à inclusão, alteração e exclusão de dados.
Isso quer dizer que as operações realizadas sobre a base de dados tratam conjuntos de informações (tabelas), não registro por registro. Isso quer dizer que a linguagem de alto nível não é procedural, uma característica das linguagens denominadas de quarta geração.

Regra número 8 - Independência de dados físicos

Os programas de aplicação permanecem inalterados sempre que quaisquer modificações são feitas nas representações de memória ou nos métodos de acesso.
Essa regra nos dá uma diretriz essencial quanto aos programas de aplicação: esses programas lidam somente com dados lógicos. Se houver qualquer modificação quanto ao local de armazenamento físico dos dados ou, ainda, uma mudança qualquer de índices de acesso, o programa de aplicação, logicamente estável, não sofrerá nenhuma paralisação nem precisará de alterações. Por exemplo, se desenvolvermos uma aplicação em um diretório de dados e, posteriormente, em consequência de uma migração ou expansão de hardware, essa base de dados for transferida para um outro diretório de dados, as aplicações desenvolvidas originalmente para tal base não serão afetadas, já que a definição de localização dos dados deve ser externa à linguagem de manipulação de dados.

Regra número 9 - Independência de dados lógicos

Os programas de aplicação também permanecem inalterados quando são feitos nas tabelas, mudanças de qualquer tipo para preservar a informação.
Aqui a regra implica, por exemplo, que a junção de duas tabelas não afeta os programas, assim como a criação de novos campos e as operações de seleção que dividem uma tabela por linhas ou colunas, desde que se preservem as chaves primárias. Se, por exemplo, criarmos um novo relacionamento entre duas entidades ou modificarmos a condição de relacionamento entre elas, em hipótese nenhuma isso afetará as aplicações existentes.

Regra número 10 - Independência de integridade

As restrições de integridade específicas de determinado banco de dados relacional devem ser definíveis na sublinguagem e armazenáveis no catálogo do sistema e não nos programas de aplicação.
Restrições de domínio e regras para validação de alguns ou todos os atributos também devem ser armazenáveis no dicionário de dados. Definições, por exemplo, do tipo de caractere de um campo – se numérico, se inteiro ou de tamanho variável – devem estar especificadas nele. Já quanto ao controle de sua validade (consistência), deve haver a possibilidade de ser realizado pela sublinguagem do SGBD, sem que, para isso, haja necessidade de que o programa de aplicação o controle. Da mesma forma, a exigência da existência de um campo deve ser informada no dicionário e também deve ser administrada por este, ficando os programas de aplicação desobrigados de tais controles.

Regra número 11 - Independência de distribuição

Um SGBD relacional tem independência de distribuição quando sua sublinguagem permite que os programas de aplicação permaneçam inalterados enquanto os dados são distribuídos, tanto na inicialização de um sistema quanto na ocorrência de uma redistribuição.
Se um sistema, por sua implementação, tiver arquivos redistribuídos em meios ou máquinas diferentes, a sublinguagem do SGBD fará o controle e a ligação com os programas de aplicação, sem que seja necessário realizar nenhuma modificação neles.

Regra número 12 - Não subversão

Se um SGBD relacional tem uma linguagem de baixo nível ou recursos de linguagem que permitam processamento em baixo nível, essa linguagem não pode ser usada para subverter ou ignorar as regras de integridade ou restrições expressas na linguagem relacional de alto nível.
Isso quer dizer que, se processarmos registro a registro, isso não poderá implicar a burla de restrições específicas dos objetos do banco de dados. Tal regra pode ser considerada extensiva à utilização de outras linguagens capazes de interagir com o banco de dados. Por exemplo, a utilização de uma linguagem de baixo nível não poderá adicionar um registro sem os atributos definidos, no dicionário, como obrigatórios para uma determinada tabela.

3.3.4. Chaves e índices

O modelo relacional emprega o conceito diferenciado entre chaves e índices.

Um índice é um recurso físico que visa otimizar a recuperação de uma informação por meio do método de acesso. Seu objetivo está relacionado com o desempenho de um sistema, ou seja, concebidos para aumentar a velocidade de recuperação dos dados, os índices devem ser criados para os campos (ou colunas) pesquisados com mais frequência. Como em tudo o que desenvolvemos, é preciso analisar os prós e os contras, já que a criação de índices pode gerar algumas desvantagens, como o tempo que se leva para construí-los; o espaço em disco utilizado para armazená-los; a demora maior para as operações de modificação no banco de dados, pois todas as mudanças têm de ser realizadas nos dados e nos índices. Esses problemas podem ser minimizados ou até mitigados com o uso de regras para a criação de **campos indexados**.

• Colunas que devem ser indexadas: chave primária; as que frequentemente são utilizadas em junção (chave estrangeira); as frequentemente pesquisadas em faixas de valores; as geralmente recuperadas de forma classificada.
• Colunas que não devem ser indexadas: as que raramente são referenciadas em uma consulta; as que contêm poucos valores únicos; as definidas com os tipos de dados text, image ou bit; quando o desempenho das atualizações é mais importante que o desempenho das consultas.

A chave designa o conceito de item de busca, ou seja, um dado que será empregado nas consultas à base de dados. É um conceito lógico da aplicação.

Podemos dizer que um campo é chave se puder ser utilizado para recuperar linhas de uma tabela. Geralmente, todos os campos de uma tabela são chaves, mas o modelo relacional preocupa-se com a definição das principais chaves de uma tabela. Assim, implementa os conceitos de chave primária e chave estrangeira, que se relacionam com duas restrições de integridade determinadas por Codd, quanto ao banco de dados. Vejamos, então, quais são os tipos de chave em um modelo relacional.

1) Chave primária

O conceito de chave primária está ligado à própria concepção do modelo relacional. Os dados estão organizados sob a forma de tabelas bidimensionais, com linhas e colunas, e o princípio nos conduz a termos uma forma de identificar uma única linha da tabela por meio de um identificador único em valor. Trata-se de um conceito fundamental para entendermos o funcionamento de um banco de dados. Quando definimos um campo como chave primária, estamos informando ao banco de dados que não pode existir mais algum registro com o mesmo valor especificado como chave primária, ou seja, os valores das chaves primárias devem ser únicos. Toda tabela relacional deve possuir esse identificador unívoco, denominado chave primária. Assim, uma tabela relacional contém um campo ou conjunto de campos concatenados, permitindo que dela se identifique uma e somente uma ocorrência. Mas em uma mesma tabela podem existir mais de um campo ou coluna com essa propriedade. A estas denominamos chaves candidatas. Entre essas candidatas a chave primária, temos de escolher uma para ser definida e utilizada como tal, deixando as demais como chaves alternativas.

A chave primária também pode ser formada pela combinação de mais de um campo, pois há casos em que não se pode atribuir essa função a um único campo, já que este pode conter dados repetidos. Nessa situação, podemos combinar dois ou mais campos para formar nossa chave primária. Quando a chave primária é composta, devemos ficar atentos ao desempenho das consultas, que é inversamente proporcional ao tamanho da chave primária. Quanto maior o tamanho da chave primária, maior será o tempo de retorno de uma consulta.

2) Chaves candidatas

Uma tabela também pode conter alternativas de identificador único, pois várias colunas ou concatenações diferentes de colunas podem ter essa propriedade e, portanto, são candidatas a chave primária. Mas, como somente uma será escolhida como chave primária, as restantes passam a ser consideradas chaves alternativas.

3) Chaves secundárias

O termo chave sempre se refere a um elemento de busca por meio do qual podemos recuperar uma informação ou um conjunto de informações. E, nas tabelas do banco de dados relacional, há colunas ou conjuntos de colunas concatenadas que nos permitem fazer não uma identificação única, mas de um grupo de linhas de uma tabela. Chamamos tais colunas ou conjuntos de colunas concatenadas de chaves secundárias. É possível que existam N linhas em uma tabela com o mesmo valor de chave secundária.

4) Chaves estrangeiras

Um dos conceitos de importância vital no contexto do modelo relacional é o das chaves estrangeiras. Para mais bem compreender do que se trata, vale analisar as duas tabelas a seguir (Funcionario e Departamento). Criar índices para chaves estrangeiras aumenta a velocidade na execução das junções e também facilita a realização dos comandos **ORDER BY** e **GROUP BY**. Quando dizemos que duas tabelas possuem colunas comuns, devemos observar que, provavelmente, em uma da tabelas a coluna é uma chave primária. Na outra tabela, a coluna comum será caracterizada, então, como o que chamamos de chave estrangeira. É, na realidade, uma referência lógica de uma tabela à outra.

O comando Order By especifica um ou mais elementos que serão usados para classificar um conjunto de resultados e facilitar a pesquisa. O Group By serve para agrupar dados recuperados conforme critérios pré-definidos.

Conclui-se, então, que havendo uma coluna Ca em uma tabela A e uma coluna Cb, com idêntico domínio, em uma tabela B, a qual foi definida como chave primária de B, então a coluna Ca é uma chave estrangeira em A, já que a tabela A contém uma outra coluna distinta de Ca, que é sua chave primária. Não há limitação para o número de chaves estrangeiras em uma tabela. Veja nas tabelas abaixo exemplos de referência lógica por chave estrangeira.

FUNCIONARIO		
Matricula	Nome	Depto
1234	Wilson Oliveira	25
5678	Lucas Sirtori	15
9191	Andréa Oliveira	10
9287	Amanda Cristina	10
9388	Priscila Oliveira	25

DEPARTAMENTO	
Numero	Nome
10	Contabilidade
25	Sistemas
15	RH

Na tabela Funcionario temos o atributo Matricula_Funcionario como chave primária e, na tabela Departamento, a chave primária é o atributo Numero_Departamento. Assim, número do departamento em Funcionario é uma chave estrangeira, uma referência lógica que estabelece o relacionamento entre as duas entidades.

3.3.4.1. Regras de integridade

As regras de integridade existem para evitar que uma determinada coluna não tenha uma relação correspondente. Em função dos conceitos de chave primária e chave estrangeira, Codd elaborou as duas regras de integridade de dados do modelo relacional, a de integridade e a de integridade referencial.

Regra de integridade de identidade

Esta restrição se refere aos valores das chaves primárias. Se a chave primária, por definição, identifica uma e somente uma ocorrência de uma tabela, então não poderá ter valor NULO porque nulo não identifica nada, representando apenas a informação desconhecida.

Regra de integridade referencial

Se determinada tabela A tem uma chave estrangeira que é chave primária de uma tabela B, então ela deve:

- **Ser igual a um valor de chave primária existente na tabela B;**
- **Ser totalmente nula. Isso significa que uma ligação lógica está desativada.**

Conclui-se que não podemos ter um valor em uma chave estrangeira de uma tabela e que esse valor não existe como chave primária em alguma ocorrência da tabela referenciada. As regras de integridade do modelo relacional representam a garantia de que as tabelas guardam informações compatíveis. São, portanto, de extrema importância para a confiabilidade das informações do banco de dados.

A manutenção de valor nulo para uma chave estrangeira significa que não existe, para aquela ocorrência, ligação lógica com a tabela referenciada.

Utilizamos a integridade referencial para garantir a integridade das informações entre as tabelas relacionadas em um banco de dados, evitando assim inconsistências e repetições desnecessárias.

3.3.4.2. Regras de conversão do modelo E-R para o modelo relacional

Um modelo Entidade e Relacionamento pode ser convertido para um modelo Relacional de banco de dados. É preciso, porém, levar em conta as seguintes regras para cada elemento do sistema:

- **Entidades:** toda entidade torna-se uma tabela carregando todos os seus atributos. Cada atributo vira um campo da tabela. A chave primária e as chaves candidatas são projetadas para não permitir ocorrências múltiplas nem admitir nulos.

- **Relacionamento 1:N:** a tabela cuja conectividade é N carrega o identificador da tabela cuja conectividade é 1 (chave estrangeira).

- **Relacionamento 1:N:** (envolvendo autorrelacionamento): a chave primária da entidade é incluída na própria entidade como chave estrangeira, gerando uma estrutura de acesso a partir dessa chave estrangeira.

- **Relacionamento 1:1:** as tabelas envolvidas nesse relacionamento carregarão o identificador da outra (uma ou outra ou ambas) conforme a conveniência do projeto (de acordo com o acesso a essas tabelas).

- **Relacionamento 1:1:** (envolvendo autorrelacionamento): chave primária da entidade é incluída na própria entidade (chave estrangeira) e cria-se uma estrutura de acesso para ela.

- **Relacionamento N:N:** o relacionamento torna-se uma tabela, carregando os identificadores das tabelas que ele relaciona e os atributos do relacionamento (se houver).

3.4. Administração e gerenciamento

Um banco de dados (ver definição de Sistema Gerenciador de Banco de Dados – SGDB – no capítulo 2.5.10) é um conjunto de objetos SQL, como tabelas, funções e triggers, e tem de ser bem administrado para que se possa tirar de seus recursos o máximo proveito e com a melhor performance possível para cada tipo de negócio. Por isso há profissionais capacitados especificamente para essa função, conhecidos no mercado como Data Base Administrator (administrador de banco de dados) ou mesmo pela sigla DBA. Tais profissionais têm de conhecer profundamente as ferramentas de administração do banco de dados para utilizá-las de maneira eficiente, pois são eles que desenvolvem e administram estratégias, procedimentos, práticas e planos para disponibilizar documentação, além de compartilhar informações e dados corporativos necessários, no momento certo e às pessoas certas, com integridade e privacidade. Veja quais são as principais funções de um DBA:

• Definir o conteúdo de informações do banco de dados

Por meio de levantamentos de informações, o profissional deve decidir que informações devem ser mantidas no banco de dados da empresa.

• Definir a estrutura de armazenamento e a estratégia de acesso

Aqui o DBA deve indicar como os dados serão armazenados e quais serão os acessos mais frequentes.

• Promover a ligação com os usuários

Isso quer dizer que é sua função garantir a disponibilidade dos dados de que os usuários precisam, preparando-os ou os auxiliando na montagem do nível de visões.

• Definir os controles de segurança e integridade

Ou seja, determinar quem tem acesso a que porções do banco de dados e criar mecanismos que evitem inconsistências na base de dados.

• Definir a estratégia de backup e recuperação

É função do DBA ainda especificar como e quando serão feitos os backup e desenvolver uma estratégia para recuperar informações em caso de danos ao banco de dados.

• Monitorar o desempenho e atender às necessidades de modificações

O DBA deve organizar o sistema de modo a obter o melhor desempenho possível para a empresa.

Figura 42
Segurança em banco de dados.

3.4.1. Segurança em banco de dados

Quando falamos de segurança em banco de dados, precisamos ter em mente um conceito muito simples: o usuário deve acessar somente os dados estritamente necessários para realizar seu trabalho – não podemos lhe fornecer nenhuma outra permissão. Por exemplo, se a função do usuário é apenas realizar consultas, não devemos permitir que possa também alterar, excluir ou inserir dados no sistema, mas tão somente que faça consultas (figura 42).

3.4.1.1. Segurança no sistema operacional

Antes de falar sobre a segurança em banco de dados, vamos abordar o tema em relação ao sistema operacional. Vamos saber, por exemplo, como e porque se deve fazer o logon no Windows ou qualquer outro sistema operacional. E também porque no Windows a conta do usuário é usada como se fosse sua identidade, já que é por meio da conta de logon desse sistema que conseguimos identificar os usuários conectados e carregar configurações personalizadas para cada um deles. Além disso, aprenderemos a criar e a administrar contas de usuários e de grupos de usuários para aumentar a segurança de nossos dados.

É muito importante que estudemos as contas de usuários e grupos de usuários, pois elas formam a base da estrutura de segurança no Windows. Para esse sistema operacional é fundamental identificar o usuário que está tentando acessar determinado recurso, como uma pasta ou impressora compartilhada. Uma vez identificado o usuário, o Windows pode determinar, com base nas permissões de acesso, qual é seu nível de acesso e se ele tem ou não permissão para acessar o recurso em questão. A identificação do usuário é feita por meio do logon, pois, para conectar, ele tem de informar ao sistema qual é sua conta de logon e digitar a respectiva senha. Feito o logon, o usuário está identificado para o Windows.

O conceito de logon está diretamente ligado ao princípio da autenticidade, pois, antes de liberar o acesso a um recurso, o Windows precisa identificar o usuário que está tentando fazer o acesso, ou seja, precisa autentificar o usuário.

3.4.1.1.1. Definição de contas de usuários

Uma conta de usuário é sua identidade para o Windows. Em outras palavras: é a maneira do programa identificar cada usuário. A partir do momento em que é possível identifica-lo por meio do logon, o sistema também consegue manter um ambiente personalizado para ele, bem como um conjunto de permissões que definem o que ele pode e não pode fazer, a que recursos tem ou não acesso e em que nível.

Se você está trabalhando em um computador isoladamente ou em uma pequena rede para a qual não foi definido um domínio e os servidores rodam o Windows 2000 Server ou Windows Server 2003 e o Active Directory, as contas de usuários são gravadas no próprio computador. Assim, cada computador terá sua lista própria de contas de usuário, que são chamadas contas locais de usuário, tradução para local user account. Os usuários de tais contas só podem fazer o logon no computador em que estas foram criadas e acessar os recursos unicamente dessa máquina. As contas locais são criadas em uma base de dados chamada base de dados local de segurança (do inglês local security database). Ao fazer o logon, o Windows compara o nome do usuário e a senha fornecida com os dados da base de segurança local. Se os dados forem reconhecidos, o logon se completa. Caso contrário, é negado e o sistema emite uma mensagem de erro.

Já em redes de grande porte, baseadas em servidores Windows Server 2003, normalmente se criam domínios. Em um domínio existe apenas uma lista de contas de usuários e grupos de usuários, a qual é compartilhada por todos os computadores que o integram. A lista de usuários é mantida nos servidores da rede, nos chamados Controladores de Domínios, ou DCs (do inglês Comain Controllers). Quando um usuário faz o logon, por meio de uma conta da lista, através da rede, o Windows verifica se ele forneceu nome e senha válidos para o domínio e só libera a conexão em caso afirmativo. Contas de domínio são armazenadas na base de segurança dos servidores do domínio em questão, a qual é conhecida como SAM no NT Server 4.0 e como Active Directory no Windows 2000 Server e no Windows Server 2003.

Há dois tipos de conta de usuário disponíveis no seu computador: administrador do computador e limitada. Existe também uma conta de convidado (guest) para usuários que não têm conta configurada no equipamento. Vejamos agora quais são as características de cada tipo de conta.

- Conta administrador

Destina-se ao usuário que pode alterar o sistema, instalar programas e acessar todos os arquivos armazenados. Este, portanto, terá permissão sobre todos os recursos do computador, inclusive as contas de todos os outros usuários. Sua única restrição é alterar o tipo de sua própria conta para conta limitada, a menos que o computador contenha um outro usuário com conta de administrador. Tal restrição visa assegurar que haja sempre pelo menos

PERMISSÕES AO USUÁRIO ADMINISTRADOR

- **Criar e excluir contas de usuário no computador.**
- **Criar senhas para as contas dos outros usuários no computador.**
- **Alterar nomes, imagens, senhas e tipos de contas dos outros usuários.**

um usuário com uma conta de administrador, ou seja alguém com permissão para instalar novos programas, configurar o Windows e alterar as contas de usuários.

- Conta limitada

Como o próprio nome sugere, destina-se ao usuário com restrições de permissão para usar os recursos do computador, como alterar a maioria das configurações ou excluir arquivos importantes. Esse usuário basicamente só pode acessar programas já instalados e alterar a imagem de sua própria conta, além de criar, alterar ou excluir sua própria senha.

As informações são o principal ativo das corporações. Assim, devemos ter muito cuidado ao definir o acesso a seus bancos de dados, fazendo permissões exclusivamente a pessoas que devem e podem acessá-los e na medida exata de suas necessidades de informação para o trabalho.

3.4.1.2. Permissões para banco de dados

Agora que já sabemos um pouco sobre como funciona a segurança em sistemas operacionais, vamos conhecer a estrutura de segurança em **bancos de dados**, que é bem similar, a ponto até de herdar algumas de suas características. Vejamos, primeiramente, alguns dos conceitos de permissões em banco de dados.

Podemos definir, por exemplo, as seguintes permissões para um banco de dados:

- **Create Table (Criar Tabela).**
- **Create View (Criar Consulta).**
- **Create SP (Criar Stored Procedure).**
- **Create Default (Criar Default).**
- **Create Rule (Criar Regra),**
- **Create Function (Criar Função).**
- **Backup DB (Backup do Banco de Dados).**
- **Backup Log (Backup do Log de transações).**

RESTRIÇÕES AO USUÁRIO DE CONTA LIMITADA

• Instalar software ou hardware (drivers)
• Alterar o nome ou o tipo de sua própria conta (essa atribuição é exclusiva do usuário com conta de administrador).

Para atribuir as permissões, utilizamos o comando GRANT, com a seguinte sintaxe:

```
GRANT { ALL | statement [ ,...n ] }

TO security_account [ ,...n ]
```

Agora vamos exemplificar a utilização do comando GRANT.

Exemplo 1

Garantir para o login SERVIDOR\wilson a permissão de criar novos bancos de dados:

```
USE Master

GRANT CREATE DATABASE TO [SERVIDOR\wilson]
```

Usamos, primeiro, o comando USE Master, para tornar o Banco de Dados Master o banco atual, pois isto é condição para que o sistema permita a criação de novos bancos de dados. Na prática: ao criar um banco de dados, o usuário precisa gravar os dados nas tabelas do Banco de Dados Master, que concentra as informações sobre todo o conteúdo de uma instância do **SQL** Server.

Structured Quary Language (Linguagem de Consulta Estruturada) criada no início da década de 1970, na IBM.

Quando o login for um usuário do domínio do Windows, o nome deve vir entre colchetes, como no exemplo: [SERVIDOR\wilson]

Podemos atribuir mais do que uma permissão ao mesmo tempo, para um ou mais logins ou usuários.

Exemplo 2

Atribuir as permissões CREATE TABLE, CREATE RULE e CREATE VIEW, para o usuário SERVIDOR\wilson no banco de dados Exemplo1.

```
USE Exemplo1

GRANT CREATE TABLE, CREATE RULE, CREATE VIEW

TO [SERVIDOR\wilson]
```

Exemplo 3

Atribuir as permissões CREATE TABLE, CREATE RULE e CREATE VIEW, para os usuários SERVIDOR\grupo1 e SERVIDOR \grupo2, no Banco de Dados Exemplo1.

```
USE Exemplo1

GRANT CREATE TABLE, CREATE RULE, CREATE VIEW

TO [SERVIDOR \grupo1], [SERVIDOR \grupo2]
```

Abaixo, listamos as principais permissões de banco de dados que o comando GRANT pode atribuir. Lembre-se de que para atribuirmos todas as permissões, empregamos a palavra ALL (todos).

CREATE DATABASE: Banco de Dados master.

CREATE DEFAULT: todos os bancos de dados.

CREATE PROCEDURE: todos os bancos de dados.

CREATE RULE: todos os bancos de dados.

CREATE TABLE: todos os bancos de dados.

CREATE VIEW: todos os bancos de dados.

BACKUP DATABASE: todos os bancos de dados.

BACKUP LOG: todos os bancos de dados.

Já as principais permissões de objetos do banco de dados são estas:

SELECT: Tabelas, views e colunas.

INSERT: Tabelas e views.

DELETE: Tabelas e views.

UPDATE: Tabelas, views e colunas.

EXECUTE: Stored Procedures.

Exemplo

Atribuir todas as permissões para o usuário SERVIDOR\andrea, no banco de dados Exemplo1.

```
USE Exemplo1
GRANT ALL
TO [SERVIDOR\andrea]
```

Já para retirar as permissões de banco de dados, devemos recorrer ao comando REVOKE. Saiba que podemos retirar mais do que uma permissão ao mesmo tempo, para um ou mais logins, como mostramos nos exemplos 2 e 3.

Mas, antes, confira a sintaxe para o comando REVOKE:

```
REVOKE { ALL | statement [ ,...n ] }

FROM security_account [ ,...n ]
```

Exemplo 1
Retirar a permissão de criar novos Bancos de Dados, atribuída para o login SERVIDOR\wilson, anteriormente.

```
USE MASTER

REVOKE CREATE DATABASE TO [SERVIDOR\wilson]
```

Exemplo 2
Retirar as permissões CREATE TABLE, CREATE RULE e CREATE VIEW, atribuídas para o usuário SERVIDOR\wilson no Banco de Dados Exemplo1.

```
USE Exemplo1
REVOKE CREATE TABLE, CREATE RULE, CREATE VIEW

TO [SERVIDOR\wilson]
```

Exemplo 3
Retirar as permissões CREATE TABLE, CREATE RULE e CREATE VIEW, atribuídas para os usuários SERVIDOR\grupo1 e SERVIDOR\grupo2, no Banco de Dados Exemplo1.

```
USE Exemplo1

REVOKE CREATE TABLE, CREATE RULE, CREATE VIEW

TO [SERVIDOR\grupo1], [SERVIDOR\grupo2]
```

Exemplo 4
Retirar todas as permissões atribuídas ao usuário SERVIDOR\andrea, no Banco de Dados Exemplo1.

```
USE Exemplo1

REVOKE ALL

TO [SERVIDOR\andrea]
```

3.4.1.3. Permissões a objetos do banco de dados

As permissões a objetos aplicam-se aos diversos elementos de um banco de dados. Em uma tabela, por exemplo, podemos ter permissão de leitura dos dados (SELECT), adição de novos registros (INSERT), exclusão de registros (DELETE) e assim por diante. As permissões a objetos são as que mais diretamente se relacionam com as atividades diárias dos usuários.

Se determinado usuário utiliza uma aplicação que acessa o banco de dados no servidor SQL Server e deve ter permissão somente de leitura de dados de uma determinada tabela, o incluiremos em um role com permissão SELECT na tabela e pronto, ele apenas poderá consultar os dados. A seguir confira quais são as principais permissões a objetos:

SELECT: Tabelas, views e colunas.

INSERT: Tabelas e views.

DELETE: Tabelas e views.

UPDATE: Tabelas, views e colunas.

EXECUTE: Stored procedures.

REFERENCES: Tabelas e colunas.

Também, para atribuir permissões de objetos do banco de dados devemos utilizar o comando GRANT. Nesse caso, a sintaxe do comando GRANT é essa:

```
GRANT
{ ALL [ PRIVILEGES ] | permission [ ,...n ] }
{
[ ( column [ ,...n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ,...n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
TO security_account [ ,...n ]
[ WITH GRANT OPTION ]
[ AS { group | role } ]
```

Como a sintaxe completa não é nada simples, vamos recorrer a alguns exemplos para ilustrar a utilização do comando GRANT. Lembre-se de que podemos atribuir mais do que uma permissão ao mesmo tempo, para um ou mais logins ou usuários, como mostra o exemplo 2.

Exemplo 1
Para garantir ao usuário SERVIDOR\wilson a permissão de selecionar novos registros e atualizar os registros existentes, na tabela Clientes do Banco de Dados Exemplo1:

```
Use Exemplo1
GRANT SELECT, UPDATE ON Clientes
TO [SERVIDOR\wilson]
```

Quando o login for um usuário do Windows, o nome deve vir entre colchetes, como no exemplo:

[SERVIDOR\wilson]

Exemplo 2
Garantir para os usuários SERVIDOR\wilson e SERVIDOR\andrea a permissão de selecionar novos registros, atualizá-los e excluí-los, na tabela Clientes do Banco de Dados Exemplo1:

```
Use Exemplo1
GRANT SELECT, UPDATE, DELETE ON Clientes
TO [SERVIDOR\wilson], [SERVIDOR\andrea]
```

Exemplo 3
Atribuir todas as permissões para o usuário SERVIDOR\andrea, na tabela Clientes do Banco de Dados Exemplo1.

```
USE Exemplo1
GRANT ALL ON Clientes
TO [SERVIDOR\andrea]
```

Observe que, mais uma vez, utilizamos a palavra ALL, para indicar todas as permissões. Para retirar as permissões de objetos do banco de dados utilizamos REVOKE. Neste caso, a sintaxe do comando REVOKE é a seguinte:

```
REVOKE [ GRANT OPTION FOR ]
{ ALL [ PRIVILEGES ] | permission [ ,...n ] }
{
[ ( column [ ,...n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ,...n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
{ TO | FROM }
security_account [ ,...n ]
[ CASCADE ]
[ AS { group | role } ]
```

Agora, acompanhe nos exemplos abaixo a utilização do comando REVOKE. Nos Exemplos 2 e 3 mostramos que se pode retirar mais do que uma permissão ao mesmo tempo, para um ou mais usuários.

Exemplo 1
Retirar a permissão UPDATE, atribuída para o usuário SERVIDOR\wilson, anteriormente.

```
USE Exemplo1

REVOKE UPDATE ON Clientes

TO [SERVIDOR\wilson]
```

Exemplo 2
Retirar as permissões SELECT, UPDATE E DELETE, atribuídas para o usuário SERVIDOR\wilson na tabela Clientes do Banco de Dados Exemplo1.

```
USE Exemplo1

REVOKE SELECT, UPDATE, DELETE ON Clientes

TO [SERVIDOR\wilson]
```

Exemplo 3
Retirar todas as permissões atribuídas ao usuário SERVIDOR\andrea, na tabela Clientes do Banco de Dados Exemplo1.

```
USE Exemplo1

REVOKE ALL ON Clientes

TO [SERVIDOR\andrea]
```

Observe que novamente recorremos à palavra ALL para indicar todas as permissões. Já para negar as permissões de objetos do banco de dados utilizamos o comando DENY. Veja qual é a sintaxe do comando DENY:

```
DENY
{ ALL [ PRIVILEGES ] | permission [ ,...n ] }
{
[ ( column [ ,...n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ,...n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
TO security_account [ ,...n ]
[ CASCADE ]
```

Compreenda melhor essa relativamente complexa sintaxe por meio de três exemplos. Os de números 2 e 3 mostram que podemos negar mais do que uma permissão ao mesmo tempo, para um ou mais usuários.

Exemplo 1
Negar permissão UPDATE, para o usuário SERVIDOR\user1, na tabela Clientes, do Banco de Dados Exemplo1.

```
USE Exemplo1
DENY UPDATE ON Clientes

TO [SERVIDOR\wilson]
```

Exemplo 2
Negar as permissões SELECT, UPDATE E DELETE, para o usuário SERVIDOR\wilson, na tabela Clientes do Banco de Dados Exemplo1.

```
USE Exemplo1

DENY SELECT, UPDATE, DELETE ON Clientes

TO [SERVIDOR\wilson]
```

Exemplo 3
Negar todas as permissões atribuídas ao usuário SERVIDOR\andrea, na tabela Clientes do Banco de Dados Exemplo1.

```
USE Exemplo1

DENY ALL ON Clientes

TO [SERVIDOR\andrea]
```

O Maintenance Plan Wizard (assistente de plano de manutenção) elabora um plano de manutenção que o Agente Microsoft SQL Server pode executar regularmente. Tais como tarefas de administração de banco de dados, backups, verificações de integridade de banco de dados, atualização de estatísticas.

3.4.2. Plano de manutenção de banco de dados

A partir de uma estratégia adequada de backup e restore, um plano de manutenção define uma série de tarefas que devem ser executadas no banco de dados para garantir o seu bom funcionamento e desempenho, bem como a disponibilidade das informações. No plano de manutenção podemos prever tanto tarefas de backup para evitar a perda de informações, quanto de verificação da integridade dos objetos do banco de dados.

É possível criar um plano de manutenção manualmente, mas a maneira mais fácil e prática é fazê-lo por meio do assistente **Maintenance Plan Wizard** (plano de manutenção assistente). Esse programa é capaz de criar e agendar a execução periódica de vários jobs. Cada job realizará uma determinada tarefa que definirmos.

Agora, veremos como se cria um plano de manutenção pelo Wizard. Acompanhe as 24 etapas do passo a passo.

1. Abra o SQL Server Management Studio (Iniciar -> Programas -> Microsoft SQL Server -> SQL Server Management Studio).

2. Na janela Object Explorer, localize a instância SERVIDOR\SQL e dê um clique no sinal +, a seu lado, para o computador exibir as opções disponíveis.

3. Entre as opções que surgirem na tela, clique no sinal +, ao lado da opção Management, para que sejam mostradas novas opções.

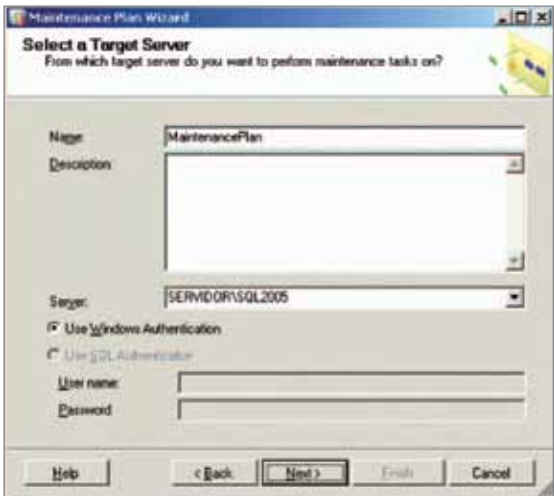



Figura 43
A tela inicial do assistente.

4. Das opções exibidas agora, clique, com o botão direito do mouse, em Maintenance Plans. No menu seguinte, clique em Maintenance Plan Wizard.

5. Surgirá agora a tela inicial do assistente, com uma mensagem informando sobre o que se pode fazer com o programa. Clique então no botão Next, seguindo para a próxima etapa. Aparecerá na tela uma página como a da figura 43.

6. Nessa etapa você definirá em qual instância do SQL Server será criado o plano de manutenção. Por padrão, a instância dentro da qual estava a opção Management -> Maintenance Plans, na qual você clicou com o botão direito do mouse, já vem selecionada. Vamos aceitar a sugestão, pois é exatamente nesta instância que queremos criar nosso plano de manutenção. Clique no botão Next, seguindo para a etapa seguinte do assistente.

7. Nessa fase você definirá quais tarefas integrarão o plano. Por padrão, todas as tarefas já vêm selecionadas. São elas: Chek database integrity, Shrink database, Defragment index(es), Re-index, Update statistics, History cleanup, Launxh SQL Server agent job, Backup database (full), Backup database (Differential) e Backup database (Transaction Log). Mantenha todas as opções marcadas, com exceção de Backup database (Differential). Clique então em Next.

8. Agora você definirá a ordem de execução das tarefas. Para alterar a ordem, marque uma determinada tarefa e depois clique no botão Move up, para movê-la para cima na lista, ou Move down, para movê-la para baixo na lista. Não vamos alterar a ordem sugerida. Assim, clique novamente em Next, e passemos à fase seguinte.

9. Abra a lista Select one or more. Aparecerão diversas opções. Nesta etapa podemos definir se o plano de manutenção incluirá todos os Bancos de Dados da Instância (All databases) ou apenas os do sistema (All system databases – master, model, and msdb). E ainda se incluirá todos os Bancos de Dados do Usuário (All user databases – all databases other than master, model, and msdb) ou apenas os selecionados (These specific databases). A opção These

TAREFAS QUE OS JOBS CRIADOS PELO DATABASE MAINTENANCE PLAN WIZARD PODEM EXECUTAR:

- **Reorganizar os dados nas páginas de dados e índices, por meio da reconstrução dos índices com um novo valor para o parâmetro Fill Factor. Isto garante melhor distribuição dos dados, com melhoria no desempenho.**
- **Compactar o banco de dados, removendo páginas de dados vazias.**
- **Atualizar uma série de informações (Indexes Statistics) sobre os índices do banco de dados.**
- **Fazer a verificação interna na consistência dos dados para certificar-se de que esses não estão corrompidos ou em estado inconsistente.**
- **Agendar o backup do banco de dados e do log de transações.**

specific databases já vem marcada por padrão. Na lista de bancos de dados selecionamos os que queremos incluir no plano de manutenção.

10. Clique em OK para fechar a lista de opções e certifique-se de que marcou a opção Include indexes, para garantir que a otimização dos índices do banco de dados AdventureWorks seja incluída no plano. Agora clique em Avançar para chegar à fase seguinte do assistente.

11. Nessa etapa, definiremos para quais bancos de dados serão criadas tarefas de otimização. Abra a lista Select one or more, que mostrará diversas opções já descritas anteriormente. A opção These specific databases já vem selecionada por padrão. Na lista de bancos de dados podemos selecionar os que desejamos incluir no plano de manutenção. Em nosso exemplo, incluiremos apenas o Banco de Dados AdventureWorks. Certifique-se, assim, de que este esteja selecionado. Clique em OK para fechar a lista de opções. Ao selecionar um banco de dados, serão habilitadas as porções de otimização e recuperação de espaço. Aceite as configurações sugeridas e clique no botão Next para seguir para a próxima etapa do assistente.

12. Agora podemos definir uma série de configurações a respeito da otimização dos índices e das páginas de dados do banco de dados. Nesta fase você tem três listas de opções. Na primeira poderá ver quais bancos de dados serão incluídos no plano de manutenção, para terem os índices de suas tabelas e views desfragmentados. Abra a primeira lista (Database(s):) e selecione somente o banco de dados AdventureWorks. Na segunda lista (Object), selecione a opção Table (para otimizar somente os índices das tabelas), a opção View (para otimizar somente os índices das views) ou a opção Tables and Views (para otimizar tanto os índices das tabelas quanto das views). Selecione agora Tables and Views. Automaticamente, serão selecionadas todas as tabelas e views. Se você selecionar uma opção individual, como por exemplo Table, na terceira lista, poderá marcar somente determinadas tabelas, para otimização de índices. Com as opções selecionadas, sua janela deve ficar semelhante à que mostramos na figura 44.

Figura 44
As opções tabelas e views.

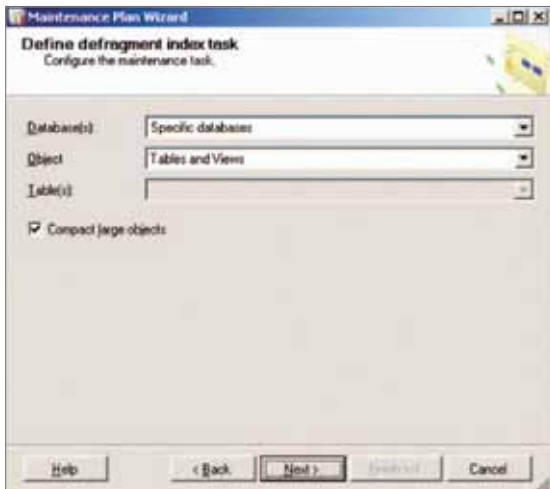



Figura 45
Aceitando as configurações padrão.

13. Clique agora no botão Next para seguir à próxima etapa do assistente.

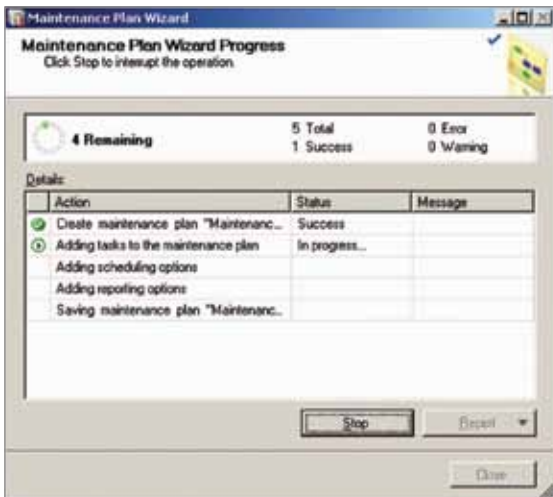
14. Nessa fase vamos escolher os bancos de dados em que serão criadas tarefas, as quais, ao serem executadas, recriarão os índices. As três primeiras listas funcionam de modo idêntico às três listas apresentadas na figura 45. Na primeira lista você marcará um ou mais bancos de dados, na segunda selecionará as opções **Table**, **Views** ou **Tables** and **Views** e, na terceira, os objetos, individualmente, dependendo de qual opção foi selecionada na segunda lista.

Esta tela tem uma série de opções avançadas, que somente um DBA experiente deverá alterar. Portanto, somente modifique essas opções se souber exatamente o que está fazendo.

Para o nosso exemplo, selecione, na primeira lista, o banco de dados AdventureWorks e, na segunda, a opção Tables and Views. Mantenha as demais opções inalteradas. E clique em Next para seguir adiante.

15. Nessa fase, você selecionará para quais bancos de dados serão criadas tarefas para atualizar as estatísticas das tabelas e views. As três primeiras listas funcionam exatamente como as três mostradas na figura 46. Marque, na primeira lista, um ou mais bancos de dados. Na segunda, selecione as opções Table, Views ou Tables and Views e, na terceira lista, os objetos, individualmente, dependendo de qual opção foi selecionada na segunda lista. Para nosso exemplo, escolha, na primeira lista, o banco de dados AdventureWorks e, na segunda, a opção Tables and Views. Mantenha as demais opções inalteradas e clique no botão Next para passar à fase seguinte do assistente.

16. Nessa etapa você selecionará quais opções de históricos serão limpas quando as tarefas do plano de manutenção (específicas para limpeza dos históricos) forem executadas e a periodicidade da execução. Você pode marcar as opções Backup and Restory history, SQL Server Agent Job history e Database Maintenance Plan History. Por padrão, as três opções já vêm selecionadas. Na parte de baixo da janela, você definirá quais dados devem ser excluídos do histórico. Por padrão também, já vem selecionada a opção 4 Week(s), significando que serão criadas tarefas, no plano de manutenção, para excluir

Figura 46
Finalizando a criação do plano de manutenção.

dados que tenham sido gravados mais de quatro semanas antes nos históricos selecionados. Aceitaremos as configurações padrão. Agora clique em Next, seguindo adiante.

17. Nessa etapa serão exibidos os jobs já existentes, criados anteriormente. Você pode marcar um ou mais jobs para serem executados, também como parte do plano de manutenção. No nosso exemplo, incluiremos todos os jobs já existentes. Portanto, certifique-se de que todos foram selecionados e clique em Next.

18. Agora você definirá para quais bancos de dados serão criadas tarefas de backup, como parte do plano de execução. Abra a lista Databases e, abaixo da opção These specific databases, marque o banco de dados AdventureWorks e clique em OK. As demais opções serão habilitadas. Você pode definir se o backup será feito em disco ou fita e em qual pasta (no caso de backup em disco), se os backups já existentes devem ser sobrescritos ou não e assim por diante. Defina as opções desejadas e clique em Next.

19. Nessa etapa você pode criar tarefas que farão o backup diferencial de um ou mais bancos de dados. Vamos incluir um backup diferencial do banco de dados AdventureWorks como parte do plano de manutenção. Na lista Database(s): selecione o banco de dados AdventureWorks e clique em OK. Aceite as demais opções e clique em Next, seguindo à fase seguinte.

20. Nessa etapa você poderá criar tarefas que farão o backup do log de transações de um ou mais bancos de dados. Vamos incluir um backup do log do banco de dados AdventureWorks, como parte do plano de manutenção. Na lista Database(s): selecione o banco de dados AdventureWorks e clique em OK. Aceite as demais opções e siga adiante, clicando em Next.

21. Agora você poderá definir uma conta conhecida como Proxy Account. Se for configurada uma conta como esta, as tarefas serão executadas no contexto desta conta, que, portanto, deverá ter todas as permissões necessárias para executar as tarefas do plano de manutenção. Não vamos configurar uma conta como Proxy Account em nosso exemplo. Então, clique novamente em Next.

22. Nessa etapa você definirá em qual pasta será gravado um relatório sobre o plano de manutenção. Por padrão, vem selecionada a pasta C:\. Aceite as configurações sugeridas e clique em Next.

23. Será exibida agora a tela final do Wizard. Se você precisar alterar alguma opção, recorra ao botão Back. Para finalizar o assistente e criar o plano de manutenção, clique em Finish. O SQL Server mostrará o progresso da criação do plano de manutenção, conforme indica a figura 46.

24. Uma vez concluída a criação do plano de manutenção, o sistema exibirá uma janela com o resultado da criação. Clique em Close para fechar a janela. Pronto, o plano de manutenção foi criado.

3.4.3. Uma linguagem versátil: SQL

Segundo Oliveira (2000), quando os bancos de dados relacionais estavam em desenvolvimento, foram criadas linguagens para manipulá-los. A SQL (sigla do inglês Structured Query Language, literalmente Linguagem de Consulta Estruturada) foi desenvolvida no início dos anos 1970, no departamento de pesquisas da IBM, como interface para o sistema de banco de dados relacional denominado SYSTEM R. Em 1986, o American National Standard Institute (ANSI) publicou um padrão de SQL e essa linguagem acabou se tornando padrão de banco de dados relacional.

A SQL apresenta uma série de comandos para definir dados, chamada de DDL (Data Definition Language ou linguagem de definição de dados) a qual é composta, entre outros, pelo Create, que se destina à tarefa de criar bancos de dados. Já os comandos da série DML (Data Manipulation Language ou linguagem de manipulação de dados), possibilitam consultas, inserções, exclusões e alterações em um ou mais registros de uma ou mais tabelas de maneira simultânea. Uma subclasse de DML, a DCL (Data Control Language ou linguagem de controle de dados) dispõe de comandos de controle, como o próprio nome diz.

Conforme Oliveira (2000), a grande virtude da linguagem SQL é sua capacidade de gerenciar índices sem a necessidade de controle individualizado de índice corrente, algo muito comum nas linguagens de manipulação de dados do tipo registro a registro. Outra característica bastante importante em SQL é sua capacidade de construção de visões, ou seja, formas de visualizar os dados em listagens independentes das tabelas e organizações lógicas dos dados.

Também é interessante na linguagem **SQL** o fato de permitir o cancelamento de uma série de atualizações ou de gravá-las, depois que iniciarmos uma sequência de atualizações. Isto pode ser feito por meio dos comandos Commit e Rollback.

É importante notar que a linguagem SQL só pode nos fornecer tais soluções, porque está baseada em bancos de dados, que garantem por si mesmo a integridade das relações existentes entre as tabelas e seus índices.

3.4.3.1. Como utilizar os comandos SQL

Segundo Oliveira (2001) a linguagem SQL foi desenvolvida para acessar os bancos de dados relacionais. Seu objetivo é fornecer um padrão de acesso aos bancos de dados, seja qual for a linguagem usada em seu desenvolvimento. Mas, apesar da tentativa de torná-la um padrão (ANSI), cada fornecedor hoje possui uma série de extensões que deixam as várias versões incompatíveis entre si. Alguns bancos de dados suportam o padrão SQL ANSI-92, mais abrangente, o que representa um esforço para facilitar o processo de tornar transparente a base de dados utilizada pela aplicação. Entretanto, nem todos os fornecedores já oferecem suporte completo ao SQL ANSI-92 porque para isto teriam de alterar partes estruturais de seus sistemas gerenciadores.

3.4.3.2. Categorias da Linguagem SQL

De acordo com Oliveira (2001), as instruções SQL podem ser agrupadas em três grandes categorias:

- **DDL (Declarações de Definição de Dados):** parte da linguagem com comandos para criação de estruturas de dados como tabelas, colunas, etc. Exemplo: CREATE TABLE.

- **DML (Declarações de Manipulação de Dados):** parte da linguagem com comandos para acessar e alterar os dados armazenados no banco de dados. Os principais comandos dessa categoria são: SELECT, UPDATE, INSERT e DELETE.

- **DCL (Declarações de Controle de Dados):** parte da linguagem com comandos para definir usuários e controlar seus acessos aos dados. Exemplo: GRANT.

3.4.3.2.1. Instruções SQL

• Instrução CREATE DATABASE

De acordo com Oliveira (2000), para gerenciar os bancos de dados com comandos SQL é necessário que se esteja posicionado no banco de dados Master. Podemos criar um banco de dados com o comando SQL, CREATE DATABASE. A sintaxe completa é:

```
CREATE DATABASE nome_bancodedados
[ON {
[PRIMARY] (NAME = nome_logico_arquivo,
          FILENAME = 'caminho_e_nome_arquivo'
          [, SIZE = tamanho]
          [, MAXSIZE = tamanho_maximo]
          [, FILEGROWTH = taxa_crescimento]
          } [, ... n]
     [LOG ON
          {
          (NAME = nome_logico_arquivo.
          FILENAME = 'caminho_e_nome_arquivo'
          [, SIZE = tamanho])
          } [, ..n]
     ]
```

Onde:

- **Nome_bancodedados:** é o nome do banco de dados que se deseja criar.

- **Nome_logico_arquivo:** é um nome usado para referenciar o arquivo em quaisquer comandos SQL executados depois que o banco de dados tiver sido criado.

- **PRIMARY:** especifica o grupo de arquivos primwário. Esse grupo deve conter todas as tabelas de sistema para o banco de dados. Um banco de dados só pode ter um grupo de arquivo PRIMARY. Se não for especificado algum, o primeiro listado será o primário.

- **FILENAME:** aqui se deve especificar o caminho e o nome do arquivo que estamos criando. O arquivo deve necessariamente estar na mesma máquina que o servidor SQL, mas pode estar em uma unidade de disco diferente.

- **SIZE:** especifica o tamanho em megabytes que queremos alocar para o banco de dados. O valor mínimo é de 1MB, mas o padrão é 3MB para arquivos de dados e 1MB para arquivos de log.

- **MAXSIZE:** permite especificar o tamanho máximo que o arquivo pode atingir. O padrão possibilita que o arquivo cresça até que o disco fique cheio.

- **FILEGROWTH:** especifica a taxa de crescimento do arquivo. Esse ajuste não pode exceder a configuração de MAXSIZE. Um valor zero indica que não é permitido aumento. O padrão é 10%, significando que cada vez que o arquivo cresce, será alocado um espaço adicional de 10% para ele. Um

banco de dados que estiver em mais de um arquivo só é expandido depois que o último arquivo se completar.

- **LOG ON:** se aplicam aqui as mesmas definições mostradas anteriormente, exceto pelo fato de que se criará o arquivo de log de transações, e não o arquivo de dados.

• Sintaxe simplificada:

CREATE DATABASE [IF NOT EXISTS] nome_banco_de_dados

Atenção: se não especificarmos IF NOT EXISTS, e o banco de dados já existir, ocorrerá um erro.

Exemplo utilizando o MYADMIN:

No exemplo criaremos um banco de dados com o nome banco_teste, como mostra a figura 47: CREATE DATABASE banco_teste

Digitamos a instrução e clicamos no botão executar, para obter o retorno do myadmin apresentado na figura 48.

Figura 47
Criando um banco de dados.

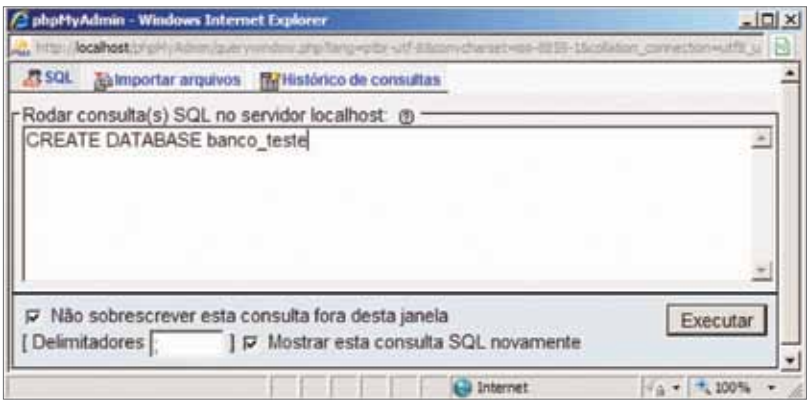



Figura 48
Retorno do myadmin.

• Instrução CREATE TABLE

Como sugere o nome, é usada para criar tabelas. Atenção para sintaxe:

CREATE TABLE Nome_Tabela

(Nom_Coluna_1 Tipo [CONSTRAINT PRIMARY| KEY| NOT NULL]

Nom_Coluna_n Tipo [CONSTRAINT PRIMARY| KEY| NOT NULL])

Exemplo:
CREATE TABLE Cliente (codigo int(7), nome varchar(40), endereco varchar(40)). Observe a figura 49.

Devemos clicar no botão executar, para ter o resultado da criação da tabela que aparece na figura 50.
Podemos verificar no lado esquerdo da tela que, agora, nosso banco de dados banco_teste possui uma tabela.

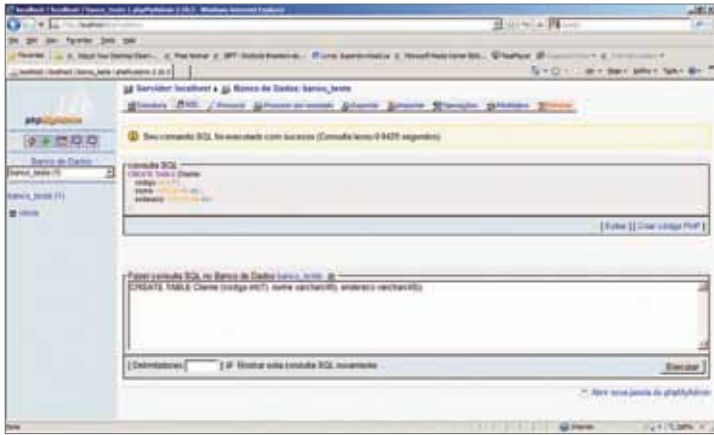
• Cláusula INSERT

O comando INSERT insere linhas em uma tabela e, sua forma mais simples, somente uma linha de dados. A sua sintaxe é:

INSERT [INTO] nome_tabela (colunas)

VALUES (valores)

Figura 49
Criando uma tabela.

Figura 50
A tabela criada.

Onde:

Nome_tabela: é o nome da tabela em que se deseja incluir os dados.

Colunas: parte da tabela onde se deseja acrescentar os dados.

Valores: é o conteúdo de cada coluna.

Exemplo no MYSQL:

INSERT INTO Cliente (codigo, nome, endereco) VALUES (123, 'WILSON', 'CAIXA POSTAL: 155 – ITU'). Como ilustra a figura 51.

Figura 51
Uso do INSERT.

Com esta instrução, iremos incluir um registro, como mostra a figura 52.

Para verificar a inclusão, utiliza-se a instrução SELECT, conforme mostra a figura 53.

Select * from cliente

E teremos o resultado mostrado na figura 54, na qual aparece listado somente o cliente Wilson.

Figura 52
Inclusão de um registro.

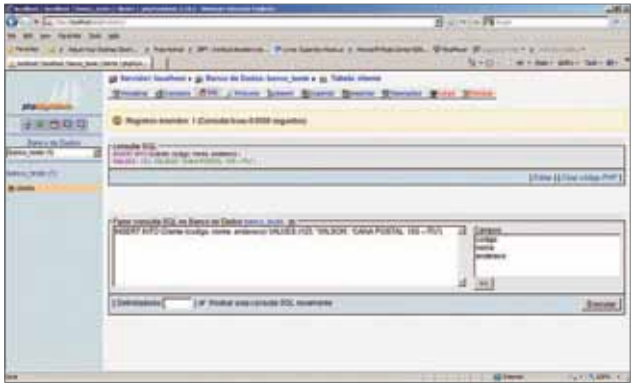



Figura 53
Verificando a inclusão.

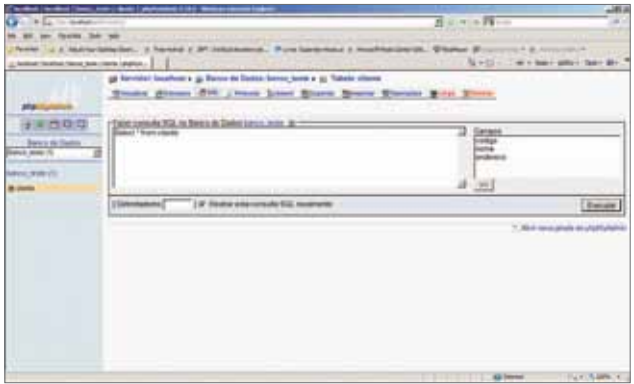



Figura 54
Registro do cliente Wilson.

• Cláusula UPDATE

Conforme Oliveira (2001), a cláusula UPDATE tem a finalidade de alterar campos de um conjunto de registros. Ou seja, para modificarmos uma ou mais linhas existentes, devemos utilizar a declaração UPDATE, cuja sintaxe é a seguinte:

```
UPDATE tabela
SET coluna=valor
Where condição
```

Onde:

Tabela: é o nome da tabela a ser atualizada.

Coluna: é o nome da coluna a ser atualizada.

Valor: é o novo valor para a coluna.

SET: determina os campos que receberão os valores.

Where: determina em quais registros a mudança ocorrerá. Na sua ausência, a mudança ocorrerá em todos os registros da tabela.

Exemplo em MYSQL:

```
UPDATE Cliente
Set nome = "Wilson Oliveira"
Where codigo= 123. (figura 55)
```

Para conferir, vamos fazer um select na tabela:

Select * from cliente. Acompanhe na figura 56.

Podemos observar na figura 57, que o nome do cliente agora aparece como Wilson Oliveira e não mais Wilson, apenas.

Figura 55
Cláusula update.

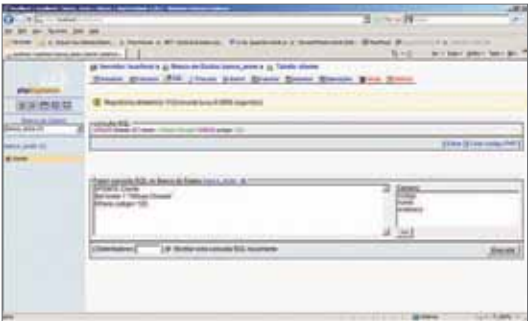



Figura 56
Select na tabela.

Figura 57
O nome do cliente completo na tabela.

• Cláusula DELETE

Tem a função de eliminar registros de uma tabela. O comando DELETE exclui permanentemente uma ou mais linhas, baseado em uma condição. Sintaxe é:

DELETE From nome_tabela Where condição

Onde:

Nome_tabela: é o nome da tabela em que se deseja excluir os dados.

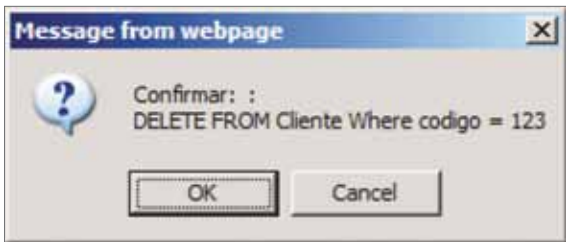
Where: determina quais registros serão eliminados da tabela.

Condição: é a condição para selecionar os dados que se deseja excluir.

DELETE FROM Cliente Where codigo = 123. Veja a figura 58**.**

Quando damos o comando DELETE, o MYSQL solicita a confirmação, para que não cometamos um erro irreparável (ou quase, pois teríamos que retornar o backup ou redigitar os dados eliminados), como ilustra a figura 59.

Figura 58
O comando DELETE.

Figura 59
Para confirmar o comando DELETE.

Se confirmarmos, teremos o retorno, conforme se apresenta na figura 60.

É preciso, então, fazer um select para verificar se o registro foi realmente eliminado (figura 61).

Podemos observar na figura 62 que, de acordo com o retorno, não há nenhum registro em nossa tabela.

Figura 60
O retorno depois do delete.

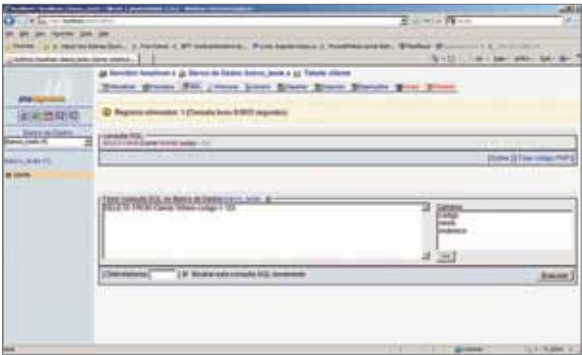



Figura 61
O SELECT para verificar o registro.

Figura 62
Sem registro na tabela.

• Comando SELECT

O Comando SELECT busca informações de um banco de dados. A sintaxe é:

```
SELECT [DISTINCT] {*, coluna [alias], ...}
FROM tabela;
```

Onde:
DISTINCT: elimina as colunas duplicadas da consulta.

*****: seleciona todas as colunas da tabela.

Coluna: especifica as colunas desejadas na pesquisa.

alias: fornece às colunas diferentes cabeçalhos.

FROM: especifica em qual tabela desejamos realizar a consulta.

Tabela: especifica que tabela será utilizada para pesquisa.

```
Exemplo:
SELECT codigo, nome
FROM departamento;
```

Exemplo com alias:

```
SELECT nome "Nome", salario*12 "Salário Anual"
FROM empresas;
```

Vamos fazer algumas inclusões para melhor verificação das instruções SELECT:

```
INSERT INTO Cliente (codigo, nome, endereco) VALUES (123,
        'WILSON OLIVEIRA', 'CAIXA POSTAL: 155 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (145,
'ANDREA SIRTORI OLIVEIRA', 'CAIXA POSTAL: 135 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (567, 'LU
        CAS OLIVEIRA', 'CAIXA POSTAL: 111 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (345,
        JOSE FRANCISCO', 'CAIXA POSTAL: 45 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (777,
        'AMANDA SIRTORI', 'CAIXA POSTAL: 233 – ITU');
```

Figura 63
O comando SELECT.

Observe a figura 63.

O resultado da inclusão aparece na figura 64.

Agora, façamos um select para verificar os registros incluídos: Select * from cliente (figura 65). Teremos o que aparece na figura 66.

3.4.3.4. Views (Visões)

Uma visão (VIEW) é uma forma alternativa de olhar os dados de uma ou mais tabelas. Para definir uma visão, usa-se um comando SELECT, que faz uma consulta sobre as tabelas. A visão aparece depois, como se fosse uma tabela.
Uma view é como uma janela que dá acesso aos dados da tabela, mas com restrições. Tem, no entanto, uma série de vantagens:

- Uma visão pode restringir quais colunas da tabela podem ser acessadas (para leitura ou modificação), o que é útil no caso de controle de acesso.
- Uma consulta SELECT, usada muito frequentemente, pode ser criada como visão. Com isso, a cada vez que é necessário fazer uma consulta, basta selecionar dados da visão.
- Visões podem conter valores calculados ou valores de resumo, o que simplifica a operação.

Figura 64
O resultado da inclusão.

Figura 65
Verificação dos registros incluídos.

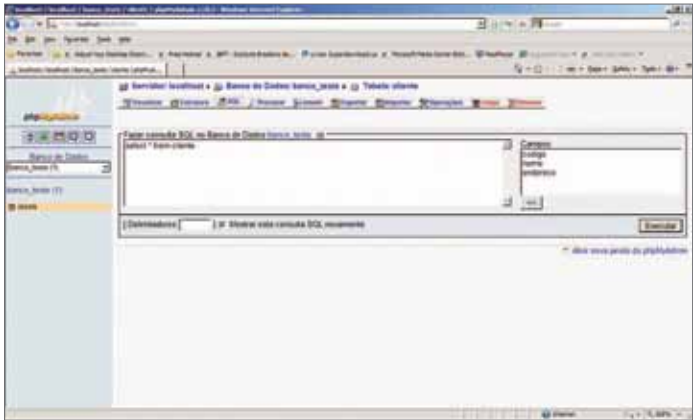



Figura 66
O resultado da verificação.

Criando uma visão com comandos SQL

Para criar uma visão através de SQL, usamos o comando CREATE VIEW. Este comando possui a seguinte sintaxe:

```
CREATE VIEW nome_visao [(coluna [,...n])]
[WITH ENCRYPTION]
AS
    Declaração_select
[WITH CHECK OPTION]
```

Onde:

Nome_visao: é o nome a ser dado à visão.

Coluna: é o nome a ser usado para uma coluna em uma visão. Nomear uma coluna em CREATE VIEW só é necessário quando uma coluna é obtida por uma

expressão aritmética, uma função, ou quando duas ou mais colunas poderiam ter o mesmo nome (frequentemente por causa de uma junção), ou ainda quando a coluna em uma visão recebe um nome diferente do nome da consulta da qual se originou. Os nomes de colunas também podem ser atribuídos no comando SELECT. Caso seja necessário nomear mais de uma coluna, entre com o nome de cada uma delas separado por vírgulas.

WITH ENCRYPTION: criptografa as entradas na tabela SYSCOMMENTS que contém o texto do comando CREATE VIEW.

WITH CHECK OPTION: força todas as modificações de dados executadas na visão a aderirem os critérios definidos na declaração SELECT. Quando uma coluna é modificada por meio de uma visão WITH CHECK OPTION garante que os dados permaneçam visíveis por meio da visão depois que as modificações forem efetivadas.

Exemplo:

A seguir, estamos criando uma VIEW que mostrará o CODIGO, NOME e SALARIO dos funcionários da tabela empregado, cujos salários são maiores que R$ 1.000,00. Esta VIEW terá o nome EMPM1000.

```
CREATE VIEW empm1000 AS
SELECT CODIGO, NOME, SALARIO
FROM EMPREGADO
WHERE SALARIO > 1000
```

Nossa VIEW de nome EMPM1000 foi criada, agora iremos dar um SELECT nela.

```
SELECT O FROM EMPM1000
```

E teremos o resultado dos funcionários com salário maior que R$ 1.000,00

Outro exemplo:

Criaremos uma VIEW chamada EMPMEDIA, baseados na tabela EMPREGADO, que mostrará o maior salário, o menor salário e a média salarial. Utilizaremos o recurso de Alias para mudar os nomes das colunas da VIEW para MAIOR, MENOR e MEDIA, conforme o exemplo seguinte:

```
CREATE VIEW EMPMEDA

(MAIOR, MENOR, MEDIA)

AS
```



```
SELECT MAX(SALARIO), MIN(SALARIO), AVG(SALARIO)

FROM EMPREGADO
```

Para vermos o resultado, devemos dar um SELECT em nossa VIEW.

```
SELECT * FROM EMPMEDIA
```

3.4.3.5. Stored Procedures (procedimento armazenado)

Um Stored Procedure (procedimento armazenado) é um conjunto de comandos SQL que são compilados e guardados no servidor. Pode ser acionado a partir de um comando SQL qualquer. Em alguns sistemas de banco de dados, armazenar procedimentos era uma maneira de pré-compilar parcialmente um plano de execução, o qual então ficava abrigado em uma tabela de sistema. A execução de um procedimento era mais eficiente que a execução de um comando SQL porque alguns gerenciadores de banco de dados não precisavam compilar um plano de execução completamente, apenas tinha que terminar a otimização do plano de armazenado para o procedimento. Além disso, o plano de execução completamente compilado para o procedimento armazenado era mantido na cache dos procedimentos, significando que execuções posteriores do procedimento armazenado poderiam usar o plano de execução pré-compilado.

A vantagem de usar procedimentos armazenados é que estes podem encapsular rotinas de uso frequente no próprio servidor, e estarão disponíveis para todas as aplicações. Parte da lógica do sistema pode ser armazenada no próprio banco de dados, e não precisa ser codificada várias vezes em cada aplicação.

Criando procedimentos armazenados

Para criar um procedimento, devemos utilizar o comando CREATE PROCEDURE. Por exemplo, o procedimento demonstrado a seguir recebe um parâmetro (@nome) e mostra todos os clientes cujos nomes contenham a informação:

```
CREATE PROCEDURE BUSCACLIENTE

@nomebusca varchar(50)
As

SELECT codcliente, Nome from CLIENTE

WHERE Nome LIKE '%'+@nomebusca+'%'
```

Devemos notar que os parâmetros são sempre declarados com @, logo após o nome do procedimento. Um procedimento pode ter zero ou mais parâmetros. Declara-se o nome do procedimento e a seguir o tipo de dados do parâmetro. Ao invés de CREATE PROCEDURE, pode-se utilizar CREATE PROC, com o mesmo efeito.

Dentro do procedimento pode haver vários comandos SELECT e o seu resultado será do procedimento. O corpo do procedimento começa com a palavra AS e vai até o seu final.

Não se pode usar os comandos SET SHOWPLAN_TEXT, e SET SHOWPLAN_ALL dentro de um procedimento armazenado, pois esses devem ser os únicos comandos de um lote (batch).

Executando procedimentos armazenados

Para executar um procedimento usa-se o comando EXEC (ou EXECUTE). A palavra EXEC pode ser omitida, se a chamada de procedimento for o primeiro comando em um Script ou vier logo após um marcador de fim de lote (a palavra GO).

Por exemplo, podemos executar o procedimento anterior da seguinte forma:

BuscaCliente 'an'

Os resultados serão as linhas da tabela cliente onde o valor de nome contém 'an' (se existirem tais linhas).

Ao executar um procedimento, podemos informar explicitamente o nome de cada parâmetro, por exemplo:

BuscaCliente @nomeBusca = 'an'

Isso permite passar os parâmetros (se houver mais de um) fora da ordem em que eles foram definidos no procedimento.

EXEC também pode executar um procedimento em outro servidor. Para isso, a sintaxe básica é:

```
EXEC
Nome_servidor.nome_banco_de_dados.nome_procedimentos
Triggers (Gatilhos)
```

Um Gatilho (Triggers) é um tipo de Stored Procedure, que é executado automaticamente quando ocorre algum tipo de alteração numa tabela. Gatilhos disparam quando ocorre uma operação INSERT, UPDATE ou DELETE numa tabela. Geralmente são usados para reforçar restrições de integridade que não podem ser tratadas pelos recursos mais simples, como regras, defaults, restrições, a opção NOT NULL, etc. Deve-se usar defaults e restrições quando estes fornecem toda a funcionalidade necessária. Um Gatilho também pode ser empregado para calcular e armazenar valores automaticamente em outra tabela.

Exemplo de Gatilhos:

Para utilizá-los, temos que criar antes algumas tabelas que serão usadas como exemplo. A tabela notafiscal conterá os cabeçalhos de notas fiscais. A tabela item notafiscal irá conter itens relacionados com as notas fiscais. Execute o Script a seguir para criar as tabelas:

```
CREATE TABLE NOTAFISCAL

        (NumeroNota numeric(10) primary key,
        ValorTotal numeric(10,2) default(0))

GO

CREATE TABLE ITEMNOTAFISCAL

(NumeroNota numeric(10) foreign key references NotaFiscal,
        CodProduto int foreign key references Produto,
        Quantidade int not null check (Quantidade > 0),
        Primary key (NumeroNota, CodProduto)

)
```

Vamos utilizar Gatilhos para duas finalidades. Primeiro, ao excluirmos uma nota fiscal, todos os seus itens serão excluídos automaticamente. Depois, quando incluirmos um item, a coluna Valor Total será atualizada na tabela NotaFiscal.

Criando os Gatilhos

Gatilhos são sempre criados vinculados a uma determinada tabela. Se a tabela for excluída, todos os seus Gatilhos serão excluídos como consequência. Ao criarmos um Gatilho, podemos especificar em qual ou quais operações ele será acionado: INSERT, UPDATE ou DELETE.

Gatilho para inserção

Realiza a inclusão de uma ou mais linhas na tabela virtual chamada inserted, que contém as linhas que serão incluídas (mas ainda não foram). Essa tabela tem a mesma estrutura da tabela principal. Podemos consultar dados nela com o SELECT, da mesma forma que uma tabela real.

Vamos criar um Gatilho, chamado InclusaoItemNota, que será ativado por uma operação INSERT na tabela ItemNotaFiscal. Primeiro, ele vai verificar se os valores que estiverem sendo inseridos possuem uma Nota Fiscal relacionada ou não. Devemos digitar as instruções a seguir:

```
CREATE TRIGGER InclusaoItemNota
On ItemNotaFiscal for insert
As
    If not exists (select * from
    Inserted, NotaFiscal
    Where inserted.NumeroNota = NotaFiscal.NumeroNota)
    Raiserror('Esse item não contém um número de nota válido')
    Update NotaFiscal
    Set ValorTotal = ValorTotal +
(select i.Quantidade * p.preco from produto p, inserted i
Where p.codProduto = i.CodProduto)
    Where NumeroNota = (select NumeroNota from inserted)
```

Primeiramente, o Gatilho usa as tabelas inserted e NotaFiscal para consultar se existe o valor de NumeroNota na tabela. Caso não exista, o comando RAISERROR gera um erro de execução, com uma mensagem que será retornada para a aplicação. Esse comando efetivamente cancela o comando INSERT que estiver sendo executado.

Depois verifica a quantidade que está sendo inserida (inserted.Quantidade) e multiplica pelo preço do produto (Produto.Preco). Este preço é encontrado na tabela do produto usando como valor de pesquisa o código do produto (inserted.CodProduto). Ele atualiza a nota fiscal relacionada com o item que está sendo inserido, para isso verifica Where NumeroNota=(select NumeroNota from inserted).

Gatilhos para exclusão

Na exclusão, as linhas da tabela são removidas e colocadas na tabela virtual default deleted, que tem a mesma estrutura da tabela principal. Um Gatilho para exclusão pode consultar DELETED para saber quais linhas foram excluídas.

Vamos criar um Gatilho na tabela NotaFiscal para que, quando a nota fiscal for excluída, todos os seus itens de nota relacionados na tabela ItemNotaFiscal, sejam excluídos em cascata. Para isso, devemos utilizar a seguinte sequência:


```
CREATE TRIGGER ExclusaoNota
On NotaFiscal for delete
As
Delete from ItemNotaFiscal
Where NumeroNota in (select NumeroNota from deleted)
```

Gatilhos para atualização

As tabelas INSERTED e DELETED, como já vimos, são tabelas virtuais que podem ser usadas dentro de um Gatilho. A primeira contém os dados que estão sendo inseridos na tabela real e a segunda, os dados antigos, que estão sendo incluídos.

Num Gatilho de atualização (FOR UPDATE), essas duas tabelas também estão disponíveis. No caso, DELETED permite acessar os dados como eram antes da modificação e INSERTED dá acesso aos dados depois da atualização.

Podemos então considerar uma atualização como uma exclusão seguida de uma inserção (excluem-se valores antigos e inserem-se valores novos).

Por exemplo, ao mudar a quantidade em um ItemNotaFiscal, o total da Nota deve ser recalculado. Para isso é levada em conta a diferença entre a quantidade antiga (deleted.Quantidade) e a nova (inserted.Quantidade). Vamos agora criar um Gatilho em ItemNotaFiscal que faz isso:

```
CREATE TRIGGER AlteracaoItemNota
On ItemNotaFiscal for update
        As
        If update(Quantidade) or update(CodProduto)
        Begin
          Update NotaFiscal
          Set ValorTotal = ValorTotal +
          (select p.preco * (i.Quantidade – d.Quantidade)
          From Produto p inner join inserted i
```

```
On p.CodProduto = i.CodProduto inner join deleted  d

On i.CodProduto = d.CodProduto and i.NumeroNota
          = d.NumeroNota)
  End
```

Note que o uso de "if update(nome_da_coluna)", dentro de um Gatilho de atualização, permite descobrir se a coluna está sendo alterada ou não. Isso para evitar trabalho desnecessário, caso não haja alguma modificação.

3.4.3.6. Um exemplo completo

Vamos construir um exemplo utilizando o phpMyAdmin com o banco de dados MySQL. Primeiro, vamos criar um banco de dados chamado ERP, onde armazenaremos as tabelas referentes ao nosso sistema de gestão empresarial.

Para criar esse banco de dados devemos utilizar a seguinte instrução:

```
CREATE DATABASE erp
```

Na figura 67, mostramos como será a instrução utilizando o phpMyAdmin.

O resultado é o sucesso da instrução, como se pode observar na figura 68.

Agora, vamos criar uma tabela de clientes. Para isso, utilizaremos a seguinte instrução:

```
CREATE TABLE Cliente (codigo int(7), nome varchar(40), endereco varchar(40))
```

Na figura 69 mostramos como será a instrução utilizando o phpMyAdmin.

Podemos observar o sucesso da instrução na figura 70.

Figura 67
Usando o phpMyAdmin.

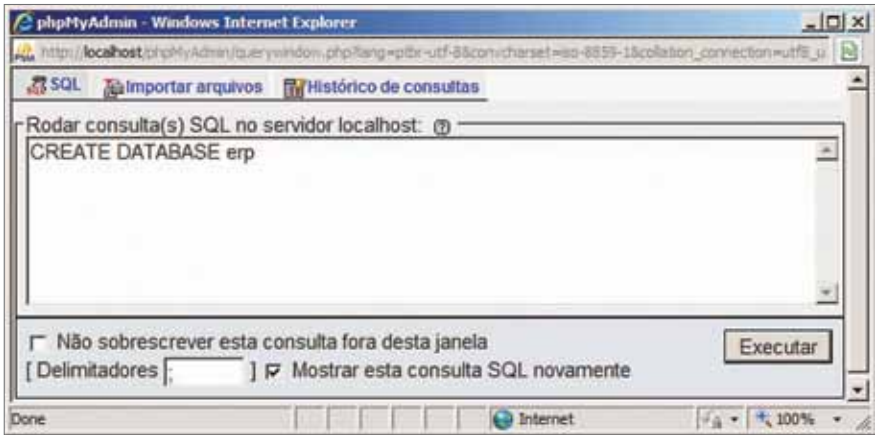



Figura 68
O resultado da instrução.

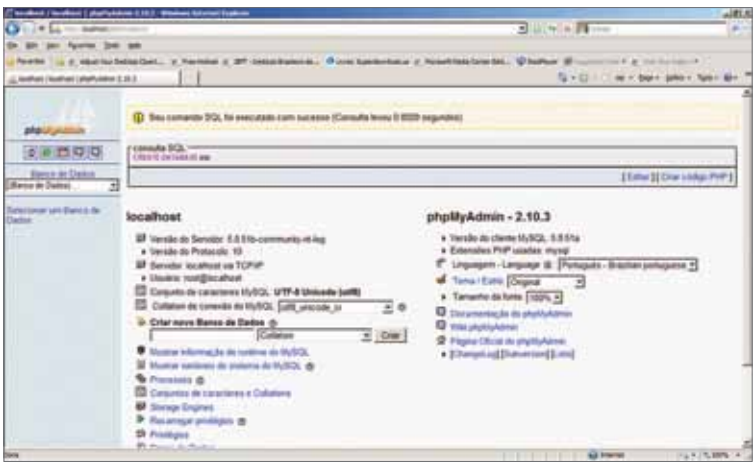
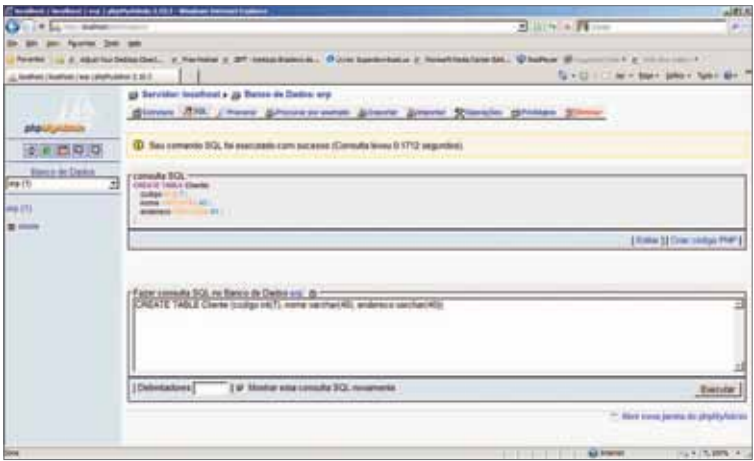



Figura 69
Instrução a partir do phpMyAdmin.

Figura 70
Instrução bem sucedida.

É hora de incluir clientes em nossa tabela para conseguirmos realizar algumas atividades. Utilizaremos as seguintes instruções INSERT para incluir cinco registros:

```
INSERT INTO Cliente (codigo, nome, endereco) VALUES (123, 'WILSON OLIVEIRA', 'CAIXA POSTAL: 155 – ITU');
```

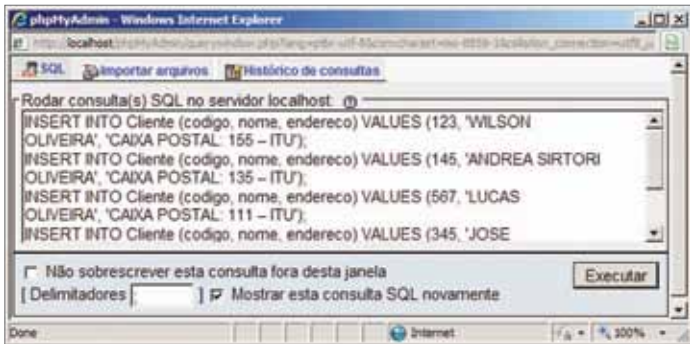
INSERT INTO Cliente (codigo, nome, endereco) VALUES (145, 'ANDREA SIRTORI OLIVEIRA', 'CAIXA POSTAL: 135 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (567, 'LUCAS OLIVEIRA', 'CAIXA POSTAL: 111 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (345, 'JOSE FRANCISCO', 'CAIXA POSTAL: 45 – ITU');

INSERT INTO Cliente (codigo, nome, endereco) VALUES (777, 'AMANDA SIRTORI', 'CAIXA POSTAL: 233 – ITU');

Veja na figura 71, como será a instrução utilizando o phpMyAdmin.

Figura 71
Instrução usando phpMyAdmin.

Podemos observar na figura 72 o sucesso da instrução.

Vamos agora fazer algumas manipulações de dados com os registros feitos na tabela de clientes. Listaremos todos esses registros, aplicando, para isto, a seguinte instrução:

```
Select * from Cliente.
```

Figura 72
Sucesso da instrução.

Figura 73
Instrução a partir do phpMyAdmin.

Observe na figura 73 como será a instrução a partir do phpMyAdmin.

O sucesso da instrução pode ser conferido na figura 74.

Figura 74
Resultado da instrução.

Vamos agora listar os clientes com código menor que 200. A instrução para isto é a demonstrada a seguir:

```
Select * from cliente where codigo < 200
```

Na figura 75, podemos observar como será a instrução empregando-se o phpMyAdmin.

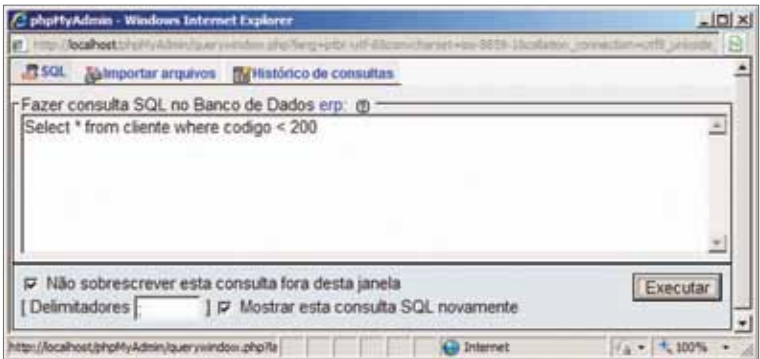
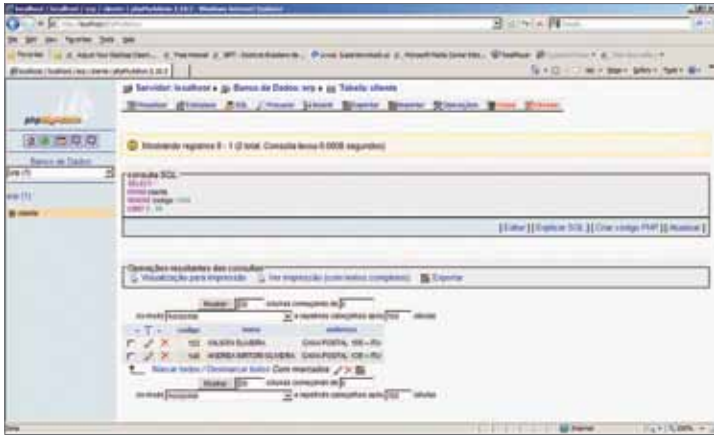



Figura 75
Listagem de clientes com código menor que 200.

Figura 76
Instrução
bem sucedida.

A figura 76 demonstra o sucesso da instrução.

Vamos alterar o cliente de código 123, de nome Wilson Oliveira, para o nome completo dele, que é Wilson Jose de Oliveira. Para isso, utilizaremos a instrução a seguir:

```
UPDATE Cliente
Set nome = "Wilson Jose de Oliveira"
Where codigo= 123
```

Na figura 77, indicamos como será a instrução, novamente aplicando-se o phpMyAdmin.

Podemos constatar o sucesso da instrução na figura 78.

Vamos listar o cliente de código 123 para verificar se o UPDATE foi bem-sucedido. Para isso devemos utilizar esta instrução:

```
Select * from cliente where codigo = 123
```

Figura 77
Instrução para o
nome completo.

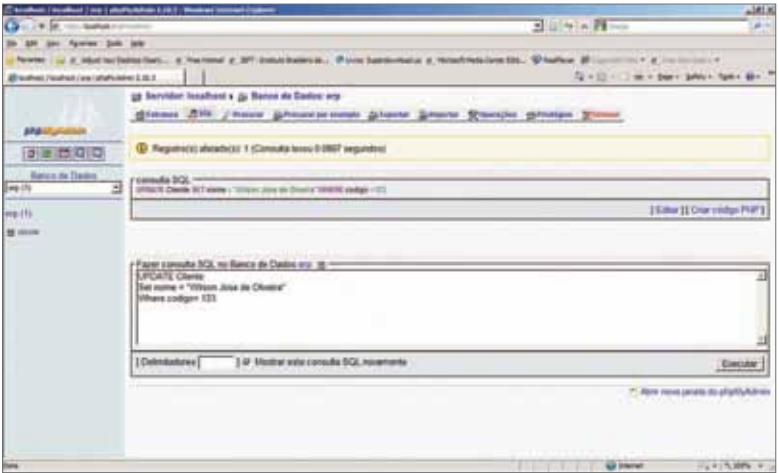
Figura 78
Outra instrução
bem sucedida.

Na figura 79, mostramos como será a instrução, outra vez utilizando o phpMyAdmin.

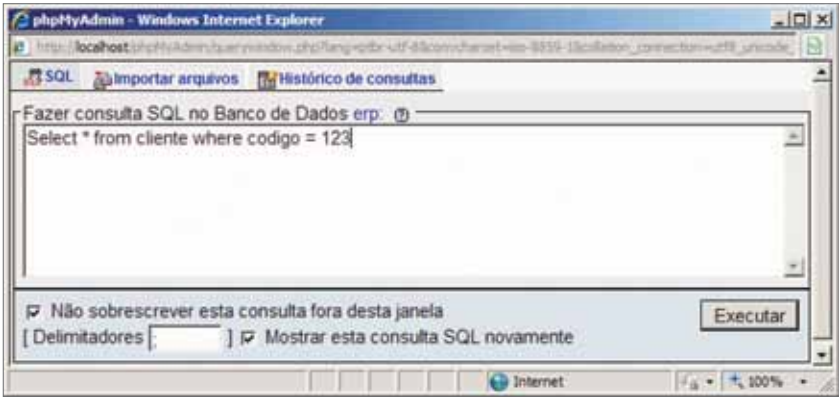
Figura 79
A instrução
para listar
cliente de
código 123.

Podemos observar na figura 80 que a alteração do cliente foi feita.

Figura 80
Realizada
a alteração
do cliente.

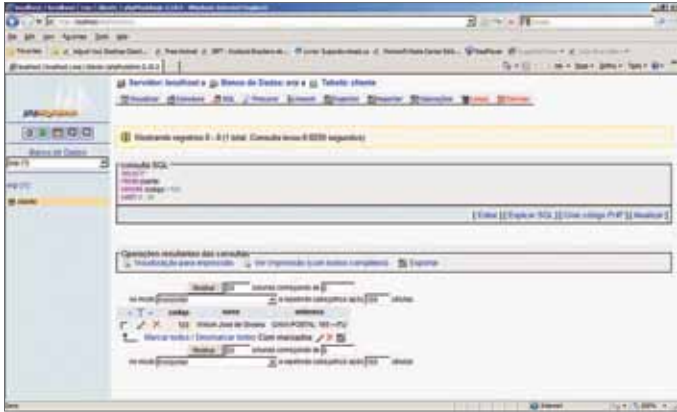



Figura 81
O comando DELETE para o cliente de código 123.

Agora vamos eliminar o cliente de código 123, utilizando a instrução DELETE, com a sintaxe apresentada a seguir:

```
DELETE FROM Cliente Where codigo = 123
```

Na figura 81, mostramos como será a instrução, usando o phpMyAdmin.

Podemos observar que a exclusão do cliente foi realizada na figura 82.

Vamos listar todos os clientes para podermos comprovar que a nossa exclusão do cliente foi bem-sucedida. Para isso, devemos utilizar a instrução SELECT, com a seguinte sintaxe:

```
Select * from cliente
```

Na figura 83, mostramos como será a instrução, aplicando o phpMyAdmin.

Podemos observar agora que o cliente de código 123 já não aparece na lista, como mostra a figura 84.

Figura 82
Efetuada a eliminação.

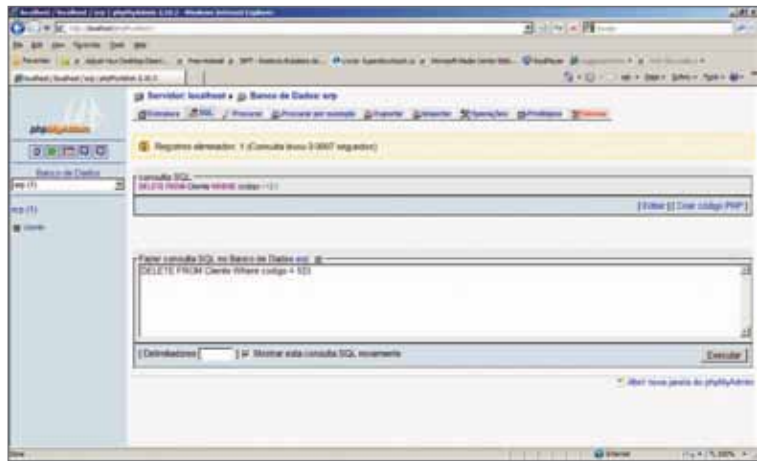



Figura 83
Listagem dos clientes para comprovar exclusão.

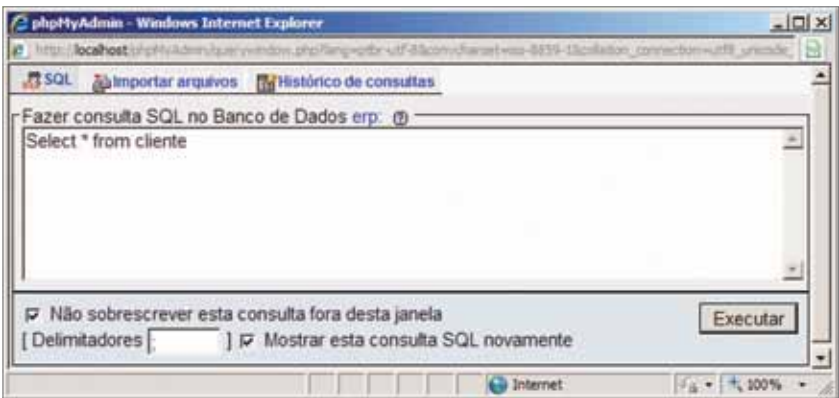



Figura 84
O cliente de código 123 foi eliminado.

Para aprender mais

A linguagem SQL é uma poderosa ferramenta de manipulação de dados. Até agora, aprendemos bastante sobre ela, mas há muito mais que você precisa saber. Para ter um conhecimento mais amplo, pesquise a linguagem mais detalhadamente e analise, por exemplo, instruções específicas para cada banco de dados disponível no mercado, como, por exemplo, SQL Server, Oracle, MySQL e Cache.

Capítulo 4

Linguagem unificada de modelagem (UML)

- Orientação a objetos
- As várias opções da UML
- O diagrama da UML
- Exemplo de desenvolvimento de projetos utilizando UML
- Considerações finais
- Referências bibliográficas

Nosso objetivo nesse capítulo é apresentar a UML (Linguagem Unificada de Modelagem), que possibilita visualização, especificação, construção e documentação de artefatos de um sistema complexo de software - o software orientado a objetos. Ou seja, vamos estudar a linguagem de definição desses softwares. Começaremos por um rápido histórico, mas só entraremos propriamente no estudo da UML após vermos os principais conceitos do paradigma orientado a objetos. Além dos conceitos mais relevantes do modelo, mostraremos, sempre que for possível, sua representação na UML, para que você vá se acostumando à linguagem.

Quando vários autores propunham metodologias para o desenvolvimento de software orientado a objetos, três estudiosos se juntaram e criaram uma linguagem unificada de modelagem, a UML. Estamos falando dos norte-americanos Grady Booch, e James Rumbaugh e do suíço Ivar Jacobson, que, em 1995, lançaram a UML 0.8 , unificando os respectivos métodos, os quais, na verdade, estavam confluindo naturalmente:

- Método de Booch, desenvolvido por Grady Booch, da Rational Software Corporation, expressivo principalmente nas fases de projeto e construção de sistemas;
- OOSE (Object-Oriented Software Engineering), de Ivar Jacobson, que fornecia excelente suporte para casos de usos como forma de controlar a captura de requisitos, a análise e o projeto de alto nível;
- OMT (Object Modeling Technique), de James Rumbaught, que era mais útil para análise e sistemas de informações com o uso intensivo de dados.

Mas podemos dizer que essa história começou bem antes, nos anos 1960, com Alan Curtis Kay, na Universidade de Utah, Estados Unidos. Considerado um dos pais da orientação a objetos com sua "analogia algébrico-biológica", Kay postulou que o computador ideal deveria funcionar como um organismo vivo, isto é, cada célula, mesmo autônoma, deveria se relacionar com outras a fim de alcançar um objetivo. As células poderiam também se reagrupar para resolver outros problemas ou desempenhar outras funções. Kay, que era pesquisador da Xerox quando surgiu a linguagem Smalltalk, no centro de pesquisas da empresa, em 1970, foi o primeiro a usar o termo orientação a objetos.

Por oferecer ferramentas que podem ser utilizadas em todas as fases do desenvolvimento de software, a UML acabou se tornando padrão de desenvolvimento de software orientado a objetos.

4.1. Orientação a objetos

A orientação a objetos é um projeto antigo na área de informática e traz consigo a idéia de aproximar o desenvolvimento de software do mundo real, criando elementos que tenham dados e funcionalidades em si mesmos. Mas sua implementação plena ainda está por vir, pois ainda hoje são várias as dificuldades no

Origem e evolução da UML

1962-1963

DIVULGAÇÃO

Ivan Sutherland

1962 - Krysten Nygaard e Ole-Johan Dahl, pesquisadores do Centro de Computação da Noruega, em Oslo, criam a linguagem Simula, derivada do Algol, introduzindo os primeiros conceitos de orientação a objetos.

1963 - Ivan Sutherland desenvolve, no MIT (Massachusets Institute of Tecnology), nos Estados Unidos, o Sketchpad, primeiro editor gráfico para desenhos orientado a objetos. O Sketchpad usava conceitos de instância e herança.

DIVULGAÇÃO

Krysten Nygaard

1969

Alan Curtis Kay apresenta sua tese de doutorado intitulada *The Reative Engine* na Universidade de Utah, na qual propõe uma linguagem (Flex), que contém conceitos de orientação a objetos.

AP PHOTO/IMAGEPLUS

Alan Curtis Kay

1970-1983

É lançada a linguagem de programação Smalltalk, desenvolvida no centro de pesquisas da Xerox, em Palo Alto, Estados Unidos. Essa foi por muito tempo a única linguagem 100% orientada a objetos.

1980 - Surge a linguagem de programação C++, que também pode ser orientada a objetos.

1983 - Jean Ichbiah cria a linguagem ADA a pedido do Departamento de Defesa dos Estados Unidos, também orientada a objetos.

caminho Há limitações de hardware, que se relacionam a problemas de acesso e armazenamento de dados, e de software, relativas a processos do sistema operacional e a falta de sistemas gerenciadores de banco de dados orientados a objetos.

4.1.1. Abstração

É característica essencial de uma entidade, que a diferencia de todos os outros tipos de entidades. Uma abstração define uma fronteira relativa à perspectiva do observador, segundo Booch, Rumbaugh e Jacobson (2005).

Como já dissemos anteriormente, abstração é a capacidade de fixar a atenção apenas nos detalhes relevantes para a construção da solução dentro de seu escopo. Isto é, quando penso em um cliente, por exemplo, não preciso pensar em todos os atributos de um cliente, mas apenas nos atributos que interessam para a solução do problema em questão. Como também vimos no capítulo 2, um cliente pode ser apenas um número.

4.1.2. Classe

De acordo com Booch, Rumbaugh e Jacobson (2005), classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica. A representação completa de uma classe tem quatro divisões, conforme conceituamos e mostramos na figura 85.

Nome da classe – Cada classe deve ter um nome que a diferencie das outras classes (BOOCH, RUMBAUGH e JACOBSON, 2005).
Atributo – É uma propriedade nomeada de uma classe, que descreve um intervalo de valores que as instâncias da propriedade podem apresentar (BOOCH, RUMBAUGH e JACOBSON, 2005).

Carro ← Nome da classe

- nomeFabricante : String
- nomeModelo : String
- cor : String
- numeroPortas : int
- anoFabricação : int
- velocidadeMaxima : double

← Atributos

+ abrirPortas() : void
+ fecharPortas() : void
+ ligar() : void
+ desligar() : void
+ acelerar() : void
+ frear() : void
+ trocarMarcha() : void

← Métodos

Responsabilidades

- Se locomover na velocidade e direção indicados pelo usuário.
- Acelerar quando solicitado.
- Frear quando solicitado.

← Responsabilidades

Figura 85
Definição da classe segundo UML 2.

4.1.2.1. Método

É a implementação de um serviço que pode ser solicitado por um objeto da classe para modificar o seu comportamento, algo que pode ser feito com um objeto e que é compartilhado por todos os objetos dessa classe (BOOCH, RUMBAUGH e JACOBSON, 2005). Existem alguns métodos especiais em praticamente todas as classes, os quais, geralmente, não representamos nos diagramas da UML por

1985-1989

DIVULGAÇÃO

Bertrand Meyer

Bertrand Meyer lança a linguagem Eifel, orientada a objetos.

1988 - Sally Shlaer e Stepen Mellor publicam o livro *Object-Oriented Systems Analysis: Modeling the World in Data*, que propõe uma técnica para análise e projetos orientados a objeto.

1989 - É criado o OMG (Object Management Group), que aprova padrões para aplicações orientadas a objeto.

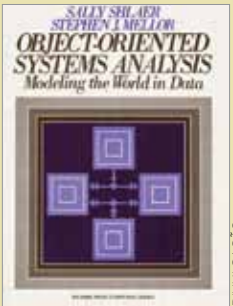
1990-1993

Peter Coad e Ed Yourdon lançam o livro *Object-Oriented Analysis*, também contendo técnicas para análise e projetos orientados a objeto.

Rebecca Wirfs-Brock, Brian Wilkerson e Lauren Wiener publicam o livro *Designing Object-Oriented Software*, tratando de técnicas de modelagem orientada a objetos.

1992 - Ivar Jacobson, Magnus Christerson, Patrik Jonsson e Gunnar Overgaard lançam o livro *Object-Oriented Software Engineering: A Use Case Driven Approach*, também sobre técnicas de orientação a objetos.
D. Embley, B. Kurtz, e S. Woodfield publicam o livro *Object-Oriented System Analysis. A Model-Driven Approach.*

1993 - Grady Booch lança seu *Booch Method* com técnicas para análise e projeto orientado a objetos.

DIVULGAÇÃO

1994-1995

James Martin e Jim Odell lançam OOIE (Object-Oriented Information Engineering).

1995 - Grupo de desenvolvedores da Sun Microsystems, da Califórnia, Estados Unidos, chefiado por James Gosling, cria a linguagem Java. James Rumbaugh publica sua OMT (Object Modeling Technique).
Os metodologistas Grady Booch, James Rumbaugh e Ivar Jacobson lançam a UML 0.8 (em 1996, surgirá a UML 0.9; em 1997, a UML 1.0; em 2000, a UML 1.4; e em 2005, a UML 2.0).

DIVULGAÇÃO

James Martin

A PRABHAKAR RAO/THE INDIA TODAY GROUP/GETTY IMAGES

James Gosling

já terem se tornado senso comum entre os desenvolvedores. Mas, sempre que você achar necessário, poderá defini-los dentro da classe.

Os métodos especiais

- **Construtor:** é o método que constrói, isto é, reserva o espaço em memória onde serão armazenadas as informações daquele objeto da classe.
- **Get:** é o método que apresenta o valor armazenado em determinado atributo de um objeto.
- **Set:** dá um valor a um atributo.

Os métodos get e set são muito úteis para preservar os atributos e garantir que sua alteração seja feita unicamente por intermédio deles. Chamamos isso de encapsulamento dos atributos de uma classe, pois podemos deixar todos eles com visibilidade privada e só manipulá-los utilizando os métodos get (para retornar o valor que está no atributo) e set (para atribuir um valor a ele). Geralmente adotamos um método set e um método get para cada **atributo da classe**.

Você deve estudar mais sobre o funcionamento e a utilização de construtores, getters e setters no livro *Programação de Computadores*, desta coleção, e nos sites que abordem a linguagem Java e linguagem C#, além de sua representação UML.

- **Destrutor:** destrói o objeto criado da memória, liberando o espaço de memória alocado na sua criação. Não é necessário criá-lo em linguagens orientadas a objetos que possuam **garbage collector**, isto é, que excluam os objetos que já não tenham referência alguma na memória.

O garbage collector procura o lixo existente na memória, ou seja, os objetos que não são utilizados e, no entanto, ocupam espaço na memória. Essa tarefa pode ser executada automaticamente ou programada.

- **Assinatura:** é a primeira linha da definição de um método, no qual podemos observar sua visibilidade, seu nome, seus parâmetros de entrada e de retorno.

Exemplo: + soma(valor1:double,valor2:double):double

A assinatura do método mostrado no exemplo acima indica que:

- o símbolo + no início da assinatura demonstra que se trata de um método público, ou seja, que todos os objetos que tiverem acesso à classe a que pertencem também poderão acessá-lo;

- o nome do método é soma;

- o método soma recebe como parâmetros de entrada dois valores do tipo double, chamados valor1 e valor2, e devolve como resultado um número do tipo **double**.

Por meio da assinatura do método podemos obter várias informações sobre sua utilização e, em alguns casos, seu funcionamento. Portanto, para facilitar a compreensão sobre o que o método faz e como devemos utilizá-lo, precisamos ter muito cuidado na definição dessas assinaturas.

4.1.2.2. Responsabilidades

São contratos ou obrigações de determinada classe. Ao criarmos uma classe, estamos criando uma declaração de que todos os seus objetos têm o mesmo tipo de estado e o mesmo tipo de comportamento (BOOCH, RUMBAUGH e JACOBSON, 2005). Dependendo do nível de detalhe (abstração) que estamos analisando no diagrama, podemos também representar graficamente uma classe apenas com seu nome ou com nome dos principais atributos e principais métodos, conforme o que queremos analisar no momento em que estamos criando o diagrama. Não precisamos, então, escrever todos os componentes da classe (figura 86).

Carro
- nomeFabricante : String
- nomeModelo : String
- cor : String
- numeroPortas : int
- anoFabricação : int
- velocidadeMaxima : double
+ abrirPortas() : void
+ fecharPortas() : void
+ ligar() : void
+ desligar() : void
+ acelerar() : void
+ frear() : void
+ trocarMarcha() : void

Carro

Figura 86
Outra forma de representar uma classe em UML 2.0.

4.1.2.3. Tipos de relacionamento entre classes

Existem basicamente três tipos de relacionamento entre classes: dependência, associação e herança.

1. Dependência: é um relacionamento de utilização, determinando que um objeto de uma classe use informações e serviços de um objeto de outra classe, mas não necessariamente o inverso. A dependência é representada graficamente por uma linha tracejada com uma seta indicando o sentido da dependência. Como você pode observar na figura 87, a classe Janela é dependente da classe Evento.

2. Associação: é um relacionamento estrutural que especifica objetos de uma classe conectados a objetos de outra classe. A partir de uma associação, conectando duas classes, você é capaz de navegar do objeto de uma classe até o objeto de outra classe e vice-versa (BOOCH, RUMBAUGH e JACOBSON, 2005). Representada por uma linha interligando as duas classes, uma associação pode definir papéis das classes relacionadas, assim como a multiplicidade de sua associação, além de ter um nome. Mas nenhum desses componentes é obrigatório em uma associação e só devem ser usados para deixar mais clara a sua definição.

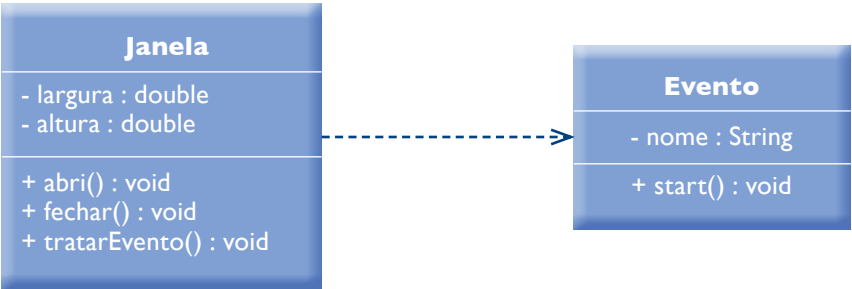
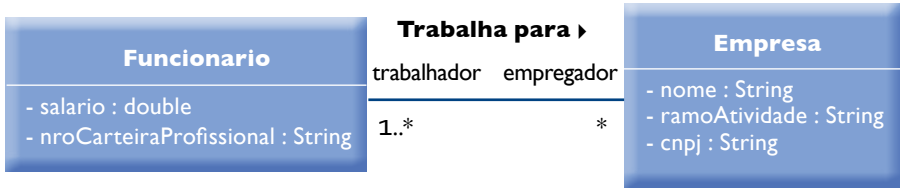



Figura 87
Representação de uma dependência em UML 2.0.

Figura 88
Exemplo de associação entre classes UML.

No diagrama da figura 88, identificamos uma associação, representada em UML 2.0 por uma linha interligando as duas classes, de nome Trabalha para, onde a classe Funcionario representa o papel de trabalhador e a classe Empresa representa o papel de empregador. Podemos observar pela representação da multiplicidade que cada objeto da classe Empresa tem, como trabalhador, 1 ou mais objetos da classe Funcionário e que um objeto da classe Funcionario tem, como empregador, no mínimo 0 e no máximo vários objetos empresa.

Já a agregação é um tipo de associação entre classes na qual é mostrada a relação todo/parte, nela uma classe fará o papel do todo e a outra, da parte. A agregação entre duas classes é representada em UML 2.0 como uma linha ligando duas classes com um losango na ponta da classe toda para diferenciar o todo da parte. Veja o exemplo na figura 89.

Observamos nesse diagrama da figura 89, que uma empresa é formada por um conjunto de departamentos, ou seja, tem a multiplicidade (leia o quadro abaixo). Vemos ainda que um departamento está associado a uma empresa.

A composição, por sua vez, é uma forma de agregação com propriedade bem definida e tempo de vida coincidente das partes pelo todo. As partes com multiplicidade não associada poderão ser criadas após a própria composição, mas, uma vez criadas, vivem e morrem com ela. Estas partes só podem ser removidas explicitamente antes da morte do elemento composto (BOOCH, RUMBAUGH e JACOBSON, 2005). Podemos dizer que numa relação de composição só faz sentido existir a parte se houver o todo. Observe o diagrama da figura 90.

Em relação à multiplicidade podemos ter:

0..* – multiplicidade de zero a muitos.
1..* – multiplicidade de 1 a muitos.
0..1 – multiplicidade de 0 a 1.
* – multiplicidade de zero a muitos; possui o mesmo significado de 0..*.
N – multiplicidade N, onde N deve ser trocado por um número que representará o número exato de objetos relacionados àquela classe.

Figura 89
À esquerda representação da agregação entre classes UML 2.0.

Figura 90
Representação de uma composição UML 2.0.

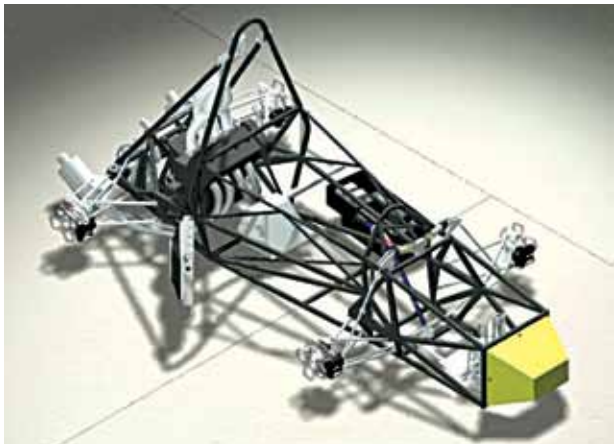
Podemos analisar que só faz sentido existirem os itens de pedido (parte) se existir o pedido (todo).

3. Herança: refere-se ao mecanismo pelo qual classes mais específicas incorporam a estrutura e o comportamento de classes mais gerais (BOOCH, RUMBAUGH e JACOBSON, 2005).

Podemos verificar, na figura 91, que a classe Pessoa possui os atributos nome e cpf e pode executar os métodos de andar e falar. Já a classe Funcionario, por herdar os atributos e métodos da classe Pessoa, possui nome, cpf, salário e nro Carteira Profissional, podendo executar os métodos andar, falar e trabalhar. Dizemos que a classe Pessoa é a superclasse da classe Funcionario, que é sua subclasse, ou que Pessoa é a classe pai de Funcionario e que Funcionario é classe filha de Pessoa.

Figura 91
Representação de herança utilizando UML.

EQUIPE EESC-USP FÓRMULA SAE

Figura 92
Classe e objeto. O "molde" que dará origem ao produto.

4.1.3. Objeto

É uma manifestação concreta de uma abstração; uma entidade com uma fronteira bem definida e uma identidade que encapsula estado e comportamento; instância de uma classe (BOOCH, RUMBAUGH e JACOBSON, 2005). Podemos imaginar uma classe como sendo o molde e os objetos, os produtos criados a partir desse molde.

Um bom exemplo é pensar nos atributos do carro como sendo: modelo, número de portas, cor, ano de fabricação, tipo de combustível, velocidade máxima. Já os métodos do carro podemos definir como: andar, parar, acelerar, entre outros.

Pensando em responsabilidades, podemos dizer que o carro tem a responsabilidade de obedecer aos comandos de seu piloto, conduzindo-o na velocidade e pelo caminho que ele escolheu, acelerando e freando de acordo com o que foi sinalizado (figura 92).

4.1.3.1. Interação entre objetos

Agora que nós já definimos e diferenciamos classes e objetos, precisamos saber como os objetos interagem entre si. De acordo com o paradigma de orientação a objetos, isso ocorre por meio de trocas de mensagens. Para entendermos como essas trocas funcionam, estudemos mais alguns conceitos.

4.1.3.2. Mensagem

Segundo Booch, Rumbaugh e Jacobson (2005), mensagens são a especificação de uma comunicação entre objetos que contêm informações à espera da atividade que acontecerá, isto é, as informações trocadas entre os objetos para que executem as funções necessárias para o funcionamento do sistema. Para seguir adiante, precisamos compreender também certas características das classes. Vejamos:

- **Encapsulamento:** Propriedade de uma classe de restringir o acesso a seus atributos e métodos. A possibilidade de definir áreas públicas e privadas em sua implementação garante mais segurança ao código criado, já que podemos definir os parâmetros de entrada e saída de um método sem revelar como ele é implementado.

- **Visibilidade:** Existem quatro tipos de visibilidade de atributos ou métodos:

  Private (privada): representada por um sinal de menos (–), é a visibilidade mais restrita e permite o acesso ao atributo ou método apenas dentro da própria classe.

  Protected (protegida): representada por um sustenido (#), permite o acesso aos métodos e atributos dentro da própria classe ou de suas subclasses.

  Public (pública): representada por um sinal de mais (+), permite o acesso aos métodos e atributos a todas as classes que tiverem algum tipo de relação com a classe em que foram criados. É a menos restritiva das visibilidades.

  Package (pacote): representada por um til (~) permite o acesso aos métodos e atributos a todas as classes incluídas no mesmo pacote.

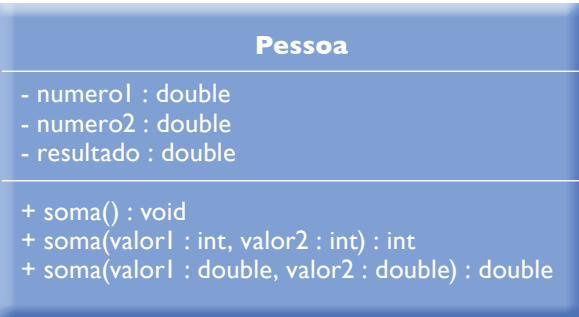
• Métodos públicos e privados

Se analisarmos a classe Pessoa, na figura 93, veremos que seus atributos são privados, mas como faremos para acessá-los? Utilizando os métodos get e set de cada atributo. É uma forma de garantir que os atributos somente serão alterados pelos métodos get e set, o que permite maior controle sobre seu uso.

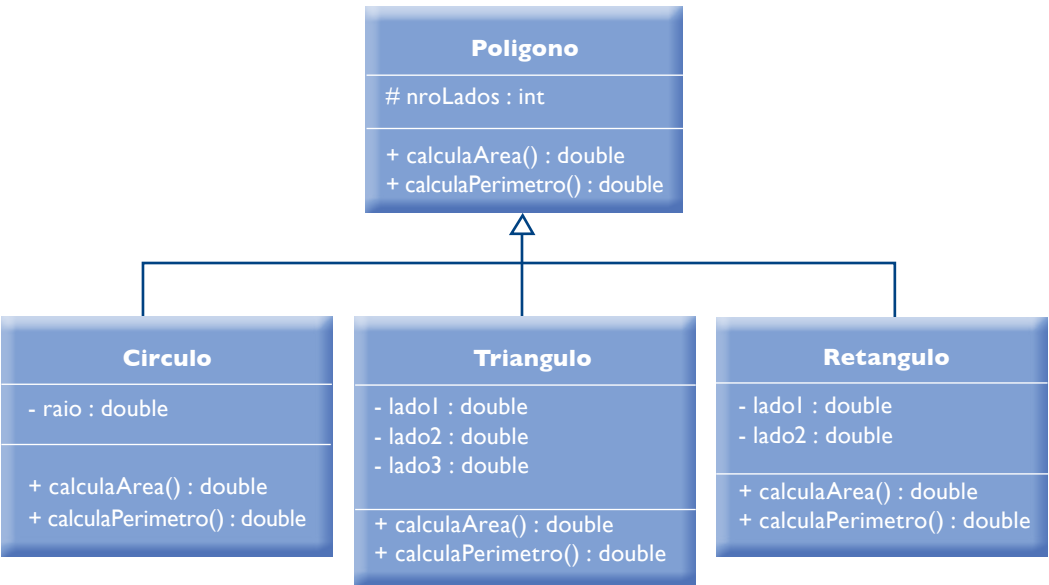
Figura 93
Exemplo de visibilidade de atributos e métodos. UML 2.0.

Figura 94
Exemplo de polimorfismo de método dito sobrecarga.

Pessoa
- numero1 : double
- numero2 : double
- resultado : double

+ soma() : void
+ soma(valor1 : int, valor2 : int) : int
+ soma(valor1 : double, valor2 : double) : double

Veja também que os métodos andar e falar são públicos, isto é, qualquer objeto que tiver acesso à classe poderá utilizá-los. Já os métodos mexerPerna e articularPalavra são privados, isto é, só podem ser utilizados dentro da classe Pessoa. Mas qual a vantagem desse tipo de implementação? Claro que para andar, uma pessoa tem que mexer a perna. O segredo do andar está no modo como ela mexe a perna. O que fizemos, então, foi garantir que o segredo, que também chamamos de regra de negócio, não faz parte da interface pública da classe, isto é, da forma com que as outras classes vão utilizá-la. Com o método encapsulado na classe, seu código está mais protegido de eventual cópia ou reprodução de como foi implementado, pois uma classe externa nem sabe de sua existência. Logo, da forma como projetamos sua implementação nenhuma outra classe saberá que para andar é necessário mexer a perna e muito menos como isso é feito. Essa é a vantagem do encapsulamento no desenvolvimento de software orientado a objetos. O mesmo princípio foi usado ao encapsular o método articularPalavra.

• **Poliformismo**: Palavra de origem grega que significa várias formas. No paradigma de orientação a objetos polimorfismo aparece em três formas diferentes:

Figura 95
Exemplo de polimorfismo sobrescrita.

- **Sobrecarga**: na mesma classe podemos ter métodos com o mesmo nome, mas que recebem parâmetros diferentes (assinaturas diferentes) e têm, por isso mesmo, funcionalidades e/ou implementações diferentes, como mostra a figura 94.

Neste exemplo da figura 94, vemos que a classe Calculadora implementa três métodos soma diferentes: o primeiro efetua a soma com os valores dos atributos numero1 e numero2, colocando o resultado no atributo resultado. Já o segundo método soma recebe como parâmetros os números inteiros valor1 e valor2 e devolve como resultado a soma dos dois. O terceiro método recebe dois valores de tipo double valor1 e valor2 e retorna como resultado a sua soma. O interpretador de comandos escolherá, em tempo de execução, com base nos tipos e no número de parâmetros informados, qual dos métodos soma será executado.

- **Sobrescrita:** em classes associadas por uma hierarquia pode haver métodos com a mesma assinatura, mas que efetuam operações diferentes. Assim, se optará pela execução de um ou de outro, dependendo da classe que estiver instanciada no momento da execução (figura 95).

Vemos, no exemplo da figura 95, que há uma superclasse chamada Poligono, que implementa os cálculos de área e perímetro para os polígonos que não sejam círculo, triângulo e retângulo, para os quais foram criadas classes filhas, de acordo com suas fórmulas. Esse é um exemplo de sobrescrita dos métodos calculaArea() e calculaPerimetro(), que, dependendo da classe a que pertence, o objeto instanciado usará o método implementado por essa classe.

- **Polimorfismo de classe**: Uma subclasse pode, dentro da aplicação, fazer o papel da superclasse (classe pai) e da subclasse (classe filha). Sempre que uma subclasse é referenciada como a superclasse, também temos polimorfismo.

• Análise e projeto de software orientado a objetos

Como já vimos no capítulo 1, o desenvolvimento de software foi dividido em fases para facilitar o processo, o que permite reduzir as questões a serem respondidas em cada etapa, e o melhor acompanhamento do projeto.

Para desenvolver softwares orientados a objetos, seguimos as mesmas fases (análise, projeto, programação, testes, implantação e manutenção) e também podemos usar as técnicas de prototipagem. Mas é necessário definir os objetivos de cada fase, principalmente a de análise e projeto orientados a objeto, que têm características um pouco diferentes das vistas até agora.

• Análise orientada a objetos

Nessa fase, fazemos levantamento e análise de requisitos, conforme as técnicas que aprendemos no capítulo 1, e definimos o que precisamos criar para satisfazer os requisitos. Resumindo: precisamos identificar quais classes deverão ser implementadas e quais serão seus principais atributos e métodos para que os requisitos sejam satisfeitos.

Utilizamos para isso, basicamente, os diagramas de casos de uso e o diagrama

de classes, que em alguns livros são chamados de diagramas de classes de análise porque as definições das classes são ainda incompletas. Isso porque nessa fase queremos apenas definir as classes e seus principais atributos e métodos e não definir em detalhes sua implementação. Em alguns casos utiliza-se também o diagrama de objetos para se poder analisar como ficariam as estruturas das classes em determinado ponto do processamento do sistema.

Todos esses diagramas serão vistos ainda nesse capítulo, mas daremos por hora uma visão de como se desenvolvem softwares orientados a objetos.

Como podemos imaginar, o produto final dessa fase são as principais classes a serem desenvolvidas para que nosso software resolva todos os requisitos definidos.

• Projeto orientado a objetos

Agora que sabemos quais são as principais classes que comporão nossa solução de software para resolver o problema proposto, precisamos definir como os objetos criados dessas classes interagirão entre si para gerar o resultado final esperado.

Nessa fase devemos projetar todo o funcionamento do software, em detalhes. Para isso podemos utilizar os demais diagramas da UML, o que nos ajudará a definir, também em detalhes, como funcionará cada um dos itens da solução, até, por exemplo, em que estrutura de hardware e software será implementada e como estarão dispostos seus diversos componentes nos computadores da rede. Geralmente, nessa fase são criadas novas classes responsáveis pela interação do usuário com o sistema, assim como com outras classes/sistemas definidos e em funcionamento. Definimos também os procedimentos de interação entre usuário e sistema, além dos requisitos de segurança de acesso às informações utilizadas pelo sistema. A UML nos oferece ferramentas que permitem analisar em detalhes cada um dos componentes de nossa solução de software nos mais diversos aspectos de sua construção e funcionamento.

Em resumo, na fase de análise orientada a objetos devemos nos preocupar com o que o sistema deve fazer para responder a todos os requisitos, definindo suas principais classes e funcionalidades. Na fase de projeto temos de pensar em como as classes deverão se comportar para que o sistema funcione de forma a cumprir todos os seus requisitos, com o tempo de resposta definido e na estrutura de software e hardware proposta.

4.2. As várias opções da UML

Ao utilizar a UML, precisamos de bom senso, para oferecer soluções adequadas e no prazo esperado pelo usuário, criando modelos apenas para as partes que realmente demandam definição mais aprofundada.

Como afirmaram seus próprios criadores, a linguagem **UML** oferece opções para análise e definição em todas as fases do desenvolvimento de software – da concepção à arquitetura de hardware e software em que a solução será implementada, passando pelo detalhamento das funcionalidades, tanto estáticas quanto dinâmicas, e fornecendo apoio à definição da melhor solução para o software orientado a objetos a ser criado.

Deve ficar claro para nós o que é estático e o que é dinâmico, na visão da UML.

Estático é aquilo que não muda dentro do software, isto é, a estrutura, a definição das classes, a modularização, as camadas e a configuração do hardware. Já a parte dinâmica diz respeito às mudanças de estado que os itens podem sofrer no decorrer da execução do software, por exemplo, pelas alterações ocasionadas pelas trocas de mensagens entre os itens nesse momento. Utilizamos a UML para criar modelos em que os diversos aspectos relevantes ao estudo e à definição da solução de software podem ser observados, para que o programa tenha qualidade e implemente as funcionalidades necessárias, com a performace esperada pelo usuário.

Já discutimos em outros capítulos as vantagens de se criar modelos no desenvolvimento de software, e a UML nos permite criá-los para todas as fases desse processo, oferecendo ao desenvolvedor subsídios para chegar a uma solução de qualidade, com uma boa visão de cada etapa.

Os autores apontam cinco diferentes visões proporcionadas pela UML durante a construção de modelos de software. São elas:

Visão de casos de uso: permite melhor compreensão do problema a ser resolvido, ajudando na definição das fronteiras do sistema, seus principais usuários e as principais funcionalidades a serem implementadas.

Visão de projeto: auxilia na análise da estrutura e das funcionalidades esperadas da solução.

Visão de processo: também chamada de visão de interação, foca o fluxo de controle entre os diversos componentes da solução, permitindo também a análise de seu desempenho, a sincronização e a concorrência entre seus componentes, necessária para o perfeito funcionamento da solução.

Visão de implementação: ajuda a definir a estrutura da solução, isto é, os arquivos de instalação, seu controle de versões.

Visão de implantação: trata da estrutura de hardware e software, ou seja, do ambiente em que a solução será implementada (figura 96).

Figura 96
Visões de um projeto utilizando UML.

4.2.1. Conceitos da estrutura da UML

Basicamente, a UML é formada por três componentes: blocos de construção, regras que restringem como os blocos de construção podem ser associados entre si e mecanismos de uso geral, que dão mais clareza às definições criadas pelos blocos de construção. Estes, por sua vez, são de três tipos: itens, relacionamentos e diagramas.

• Itens

Os itens são a base da UML, as abstrações. Já os relacionamentos são as relações entre os itens, enquanto os diagramas agrupam itens e relações, permitindo a análise de um dos aspectos da solução a ser criada. Também há diferentes tipos de itens, que são divididos em quatro grupos: estruturais, comportamentais, de agrupamento e anotacionais. Vamos estudar, a seguir, cada um dos grupos.

• Itens estruturais

Itens estruturais nos permitem definir a estrutura da solução. São formados pelas classes, as interfaces, as colaborações, os casos de uso, os componentes, os artefatos e os nós. Para começar, vamos a uma breve definição de cada um desses elementos, que mais adiante serão aprofundados, para mostrar como funcionam e explicar, por meio de um diagrama, onde são utilizados. Empregaremos também as definições já descritas no tópico sobre orientação a objetos desse livro, que deve ser consultado, caso haja necessidade de uma revisão. Apresentaremos também a notação gráfica da UML de cada componente definido.

Classes: são os elementos básicos da orientação a objetos e, consequentemente, da UML. (A classe já foi definida no tópico que trata dos conceitos de orientação a objetos, no qual você encontra inclusive sua notação na UML.)

Interfaces: são as funcionalidades a serem implementadas por uma classe ou por um componente. Demonstram o comportamento visível de uma classe, mas nunca a implementação de tal comportamento, pois contêm apenas a assinatura dos métodos, e sua implementação é feita pelas classes que herdam seu comportamento. Servem para definir comportamentos padronizados para conjuntos de classes e itens. As interfaces são representadas por um círculo, quando se trata da interface de uma classe/item, ou por um arco, no caso da interface requerida

Figura 97
As duas formas de representar uma interface.

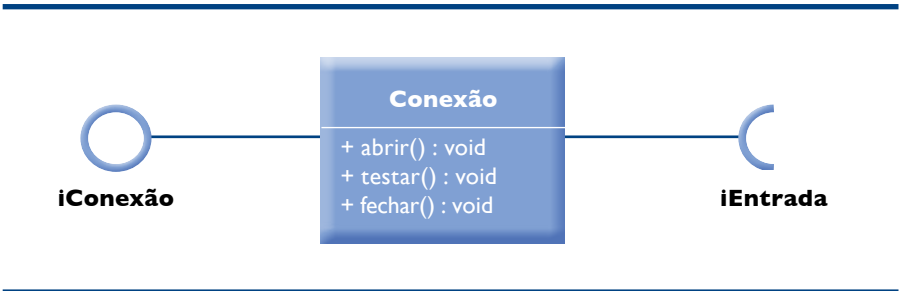
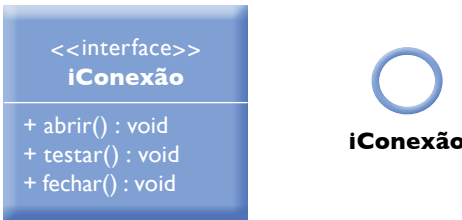



Figura 98
Classe implementando uma interface de entrada definida.

por uma classe/item. Ambas aparecem ligadas por uma linha à classe que as implementa (figura 97).

Existem, então, duas formas de representar uma interface em UML. A primeira utiliza o recurso do estereótipo <<interface>> para enfatizar que se trata de uma interface e mostra as assinaturas dos métodos que são definidos por ela. A segunda forma é a representação da interface, que não informa detalhes das funcionalidades que esta define.

Vemos, na figura 98, a representação de uma interface sendo implementada por uma classe que também tem uma definição de sua interface de entrada.

Colaboração: são agrupamentos de classes, relacionamentos e interfaces que constituem uma unidade do sistema (figura 99). Dizemos que essa unidade é maior que a soma das classes e relacionamentos implementados. São representados por uma elipse tracejada com o nome da colaboração ao centro. A colaboração serve também para nos dar uma visão em nível mais alto de abstração, quando não é necessário entrar em todos os detalhes de determinado item em análise.

Casos de uso: descrevem uma sequência de ações a serem executadas pelos componentes da solução. São ativados por um ator, servem de base para definir os comportamentos dos elementos da solução de software e são realizados por uma colaboração. São representados por uma elipse com o nome da operação que implementa no centro (figura 100).

Componentes: são estruturas que instituem uma funcionalidade de uma solução de software por meio da implementação de uma ou mais interfaces definidas. Podem ser substituídos dentro de uma solução por outro componente que implemente todas as suas interfaces, sem prejuízo ao sistema como um

Figura 99
Exemplo (à esquerda) de colaboração.

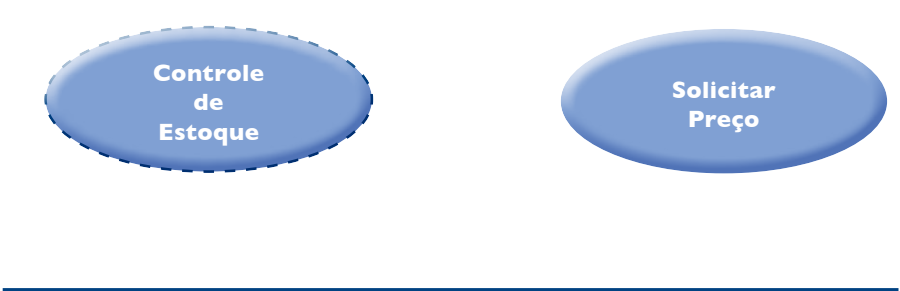



Figura 100
Exemplo de caso de uso.

Figura 101
Exemplo de componente.

Figura 102
Exemplo de artefato.

todo. São representados por um retângulo com o símbolo da UML para componentes (figura 101).

Artefato: é um elemento físico do sistema, que pode ser um programa (fonte ou executável), um script do sistema operacional e tudo o mais que pode ser transformado em bits e bytes. É representado por um retângulo com o estereótipo <<artefato>> e o seu nome (figura 102).

Nó: representa um recurso de computação. Pode ser um servidor, um computador cliente, um switch, um hub etc. Qualquer elemento computacional que faça parte da arquitetura na qual será implementada a solução pode ser definido como um nó. Em UML é desenhado como um cubo com seu nome dentro (figura 103).

Figura 103
Exemplo de nó.

• Itens comportamentais

São os itens que definem as partes dinâmicas dos modelos UML. São também chamados de verbos do modelo. Constituem itens: interações, máquina de estados e atividades.

Interações: são os conjuntos de troca de mensagens entre objetos, também chamados de comportamento. Em UML as mensagens são representadas por uma seta traçada sob seu nome (figura 104).

Máquina de estados: especifica os diversos estados pelo qual pode passar um objeto ou uma interação em seu ciclo de vida. Sua definição inclui outros componentes como estados, transições, eventos e atividades. Em UML é representada por um retângulo com os vértices arredondados (figura 105).

Figura 104
Exemplo de mensagem.

Figura 105
Exemplo de máquina de estados.

Atividade: é um comportamento que especifica a sequência de etapas realizadas por um processo computacional. É representada em UML por um retângulo de vértices arredondados (figura 106).

Figura 106
Exemplo de atividade.

• Itens de agrupamento

Servem para agrupar os demais itens da UML, ordenando-os em blocos de modo a possibilitar melhor organização do projeto. É composto apenas pelo item pacote.

Pacote: permite a inclusão de itens em seu interior para organizar o projeto, tornando-o modular e mais organizado. É conceitual, não existindo em tempo de execução. É representado por uma pasta, que pode receber apenas seu nome ou a visualização dos itens que a compõem (figura 107).

• Item anotacional

É o componente que permite a inserção de comentários nos modelos, tornando-os mais claros e inteligíveis. É composto apenas pelo item nota.

Nota: tem como objetivo inserir comentários em um modelo para deixá-lo mais compeensível. É representado por um retângulo com a ponta superior direita dobrada para dentro. Em seu interior são inseridos os comentários pertinentes ao que se quer explicar melhor dentro do modelo (figura 108). Também pode ser utilizada uma linha tracejada para apontar exatamente a que ponto do modelo se destina a explicação da nota.

Esta classe faz a conexão entre o software aplicativo e o sistema operacional

Figura 107
Exemplo de pacote.

Figura 108
Exemplo de nota.

4.2.2. Relacionamentos

Definidos os principais itens da UML, trataremos agora dos relacionamentos, que são outros blocos de construção que permitem a ligação entre os itens da UML definidos anteriormente. Existem três tipos de relacionamento em orientação a objetos: dependência, associação com seus tipos especiais (agregação e composição) e generalização (que permite a implementação do conceito de herança e realização). Todos foram apresentados quando falamos dos conceitos básicos de orientação a objetos. E reforçaremos as explicações sobre seu funcionamento quando os utilizarmos em diagramas, mais adiante.

Se você tiver alguma dúvida, volte a ler os tópicos relativos aos relacionamentos na parte de conceitos de orientação a objetos desse livro.

Assim, o próximo bloco de construção que iremos definir são os diagramas.

4.2.3. Diagramas

Existem 13 diagramas na UML 2.0, os quais são divididos em quatro grupos, de acordo com o tipo de análise que os modelos gerados por sua utilização possibilitam. São esses os grupos: diagramas estruturais, diagramas comportamentais, diagramas de interação e diagramas de implementação (figura 109).

Trataremos da construção e do uso dos diagramas implementados pela UML mais à frente, quando apresentaremos uma definição detalhada de cada um, mostrando ainda seu uso e suas principais funcionalidades. Para podermos combinar os blocos de construção da UML devemos observar as cinco regras que essa linguagem propõe, de forma que os modelos gerados contenham uma definição clara e precisa para a criação de boas soluções de software. As regras – nome, escopo, visibilidade, integridade, execução – sugerem que, ao inserir um item em um diagrama, você tem de se preocupar com cinco características que devem ficar claras à medida que cada um dos itens é inserido. As regras devem ser observadas para que possam ser criados modelos que os autores da UML chamam de bem formados, isto é, consistentes e harmônicos com todos os demais modelos que se relacionam com ele. Entenda as cinco regras:

Nome: sempre devemos lembrar que o nome de um item deve deixar claras sua formação, suas ações e responsabilidades. Não devemos nos esquecer também de que esse nome é único dentro de um modelo.

Escopo: todo item inserido em um modelo deve mostrar claramente quais são seus limites, o que implementa e quando pode ser utilizado.

Visibilidade: indica que é necessário também que fique claro quando um item estará disponível para ser utilizado e que ações estarão disponíveis por seu intermédio.

Integridade: também é importante levar em conta na criação de um item a definição clara de como este se relaciona e a consistência de tal relacionamento.

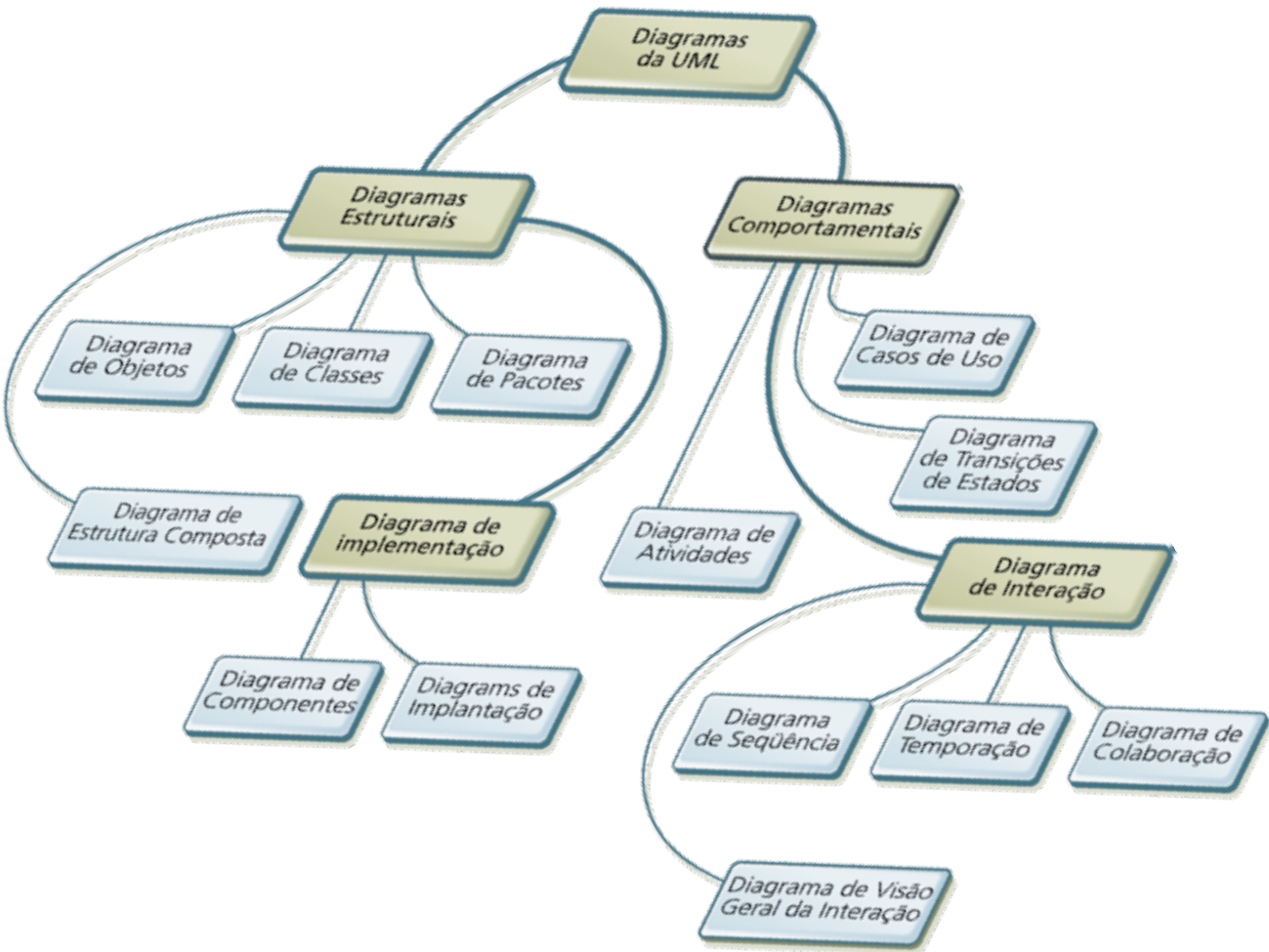



Figura 109
Diagramas definidos pela UML 2.0.

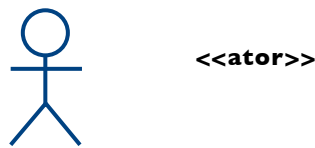
Execução: deve estar evidente ainda, o que o modelo representa e/ou simula. O que queremos observar com a criação desse modelo.

4.2.4. Adornos

Além dos três blocos de construção, a UML oferece componentes, denominados adornos, que podem ser utilizados tanto para melhorar o entendimento dos modelos criados quanto para estender o uso da UML em situações onde não existem componentes definidos. São eles:

Estereótipos: componentes de uso geral, servem para estender o significado de determinado item em um diagrama. Por serem de propósito geral, podem ser utilizados em qualquer item da UML onde for necessária uma definição mais clara de seu papel no modelo. A UML já traz uma série de estereótipos predefinidos como interface, ator, realização, mas permite que o projetista defina outros mais, sempre que surgir a necessidade. Sua representação pode ser apresentada de duas formas: uma palavra entre os símbolos de menor e maior ou um desenho do próprio estereótipo que representa. Veja o exemplo da figura 110.

Figura 110
Exemplo de estereótipo.

<<ator>>

Restrições: são utilizadas para definir regras em modelos, de forma a melhorar sua compreensão. As regras devem ser inseridas entre chaves ({}) e devem explicitar claramente a restrição. Por exemplo: {valor >=10}.

Podemos ainda criar novos compartimentos em itens da UML, tomando o cuidado de documentar claramente o que significa o novo compartimento.

A importância do profissional

A UML oferece diversos subsídios para a criação de modelos claros que nos auxiliem na construção de soluções de software de qualidade. Permite também a criação de modelos que simulam o comportamento do software em construção em diversos aspectos. Mas nunca se esqueça: sempre caberá ao desenvolvedor a responsabilidade de usar as informações de modo a obter soluções de qualidade, de acordo com as expectativas do usuário e capazes de produzir os melhores resultados possíveis.

4.3. Os diagramas da UML

Vamos agora a descrever os diagramas, seus principais componentes, a documentação envolvida. Também apresentaremos um exemplo, na medida do possível, recorreremos ao estudo de caso da padaria do senhor João (apresentada no capítulo 2 deste livro), que, você se lembra, usamos para mostrar como se cria e implementa um software em uma empresa. Por isso, recomendamos que você volte ao tema para rever as definições do problema e, assim, acompanhar com mais facilidade o próximo passo, a aplicação dos diagramas da UML.

- Todos os diagramas da UML são úteis para análise de aspectos importantes do desenvolvimento de software, mas não é necessário aplicar todos os diagramas em um mesmo projeto. Escolha apenas os quais o ajudarão a entender melhor o sistema que está desenvolvendo e a deixar mais claras as soluções que irá implementar. Nunca deixe que os diagramas fiquem grandes e complexos demais. Se perceber que isso está acontecendo, tente separá-los em mais de um, dividindo suas funcionalidades.

- Normalmente todos os diagramas são desenvolvidos com auxilio de um software.

Há até mesmo softwares que ajudam a verificar o que podemos ou não fazer em cada diagrama. Há inúmeras ferramentas que nos auxiliam nessa tarefa, muitas com versões gratuitas. Procure a que mais se adapta às suas necessidades e mãos à obra. Sugerimos que você conheça o Jude (www.jude.change-vision.com), que tem uma versão gratuita em inglês, é fácil de utilizar e permite a construção de quase todos os diagramas da UML. Escolha a que identificar como a mais adequada a seu projeto.

4.3.1 Diagrama de casos de uso

É um diagrama que mostra um conjunto de casos de uso, atores e seus relacionamentos. Abrange a visão estática de caso de uso de um sistema, conforme descrição de BOOCH, RUMBAUGH e JACOBSON (2005), *UML – Guia do usuário*, livro de uma coleção bastante útil para quem quer se aprofundar no estudo de UML (veja quadro *UML – Guia do usuário: vale a pena consultar*).

O diagrama de casos de uso é geralmente o primeiro a que recorremos no início da análise de um projeto que utilize **UML**. Ele é criado após o levantamento dos requisitos da solução imaginada – cada caso de uso é um de seus requisitos funcionais.

Ao utilizar a UML, precisamos de bom senso, para oferecer soluções adequadas e no prazo esperado pelo usuário, criando modelos apenas para as partes que realmente demandam definição mais aprofundada.

O diagrama permite visualizar os limites do sistema, sua relação com os demais sistemas, com seus componentes internos e as funções que deve realizar.

Você pode criar diagramas de caso de uso para avaliar alguma situação não

UML – Guia do usuário, vale a pena consultar

O livro *UML – Guia do usuário* integra um conjunto de obras constituído ainda pelos títulos *The Unified Modeling Language Reference Manual Second Edition* (2005) e *The Unified Software Development Process* (1999). Escritos pelos autores da linguagem – Grady Booch, James Rumbaugh e Ivar Jacobson – e editados pela Addison-Wesley, esses três livros trazem as definições e aplicações de cada um dos elementos que compõem a UML.
Os dois primeiros, que já estão na segunda edição, lançada em 2005, abordam teoria e prática, com base na versão 2.0 da UML. Já o terceiro descreve de forma completa o processo de desenvolvimento de software utilizando a linguagem.
O livro *UML – Guia do usuário* tem versão em português e, além do embasamento teórico da UML e seu uso, descreve um processo de desenvolvimento de software por meio da linguagem, exatamente como propomos aqui. Adotamos os conceitos por acreditar que, entre as mais de 50 teorias existentes sobre desenvolvimento de software, as elaboradas pelos autores do livro são as mais interessantes.

muito clara identificada nas entrevistas ou para definir como será a relação dos diversos agentes de software no sistema, ou ainda para verificar que funcionalidades este deverá implementar. O que faremos é mapear os requisitos funcionais do sistema, sua análise e também as relações que tais requisitos terão com os demais componentes, internos ou externos ao sistema.

Todo diagrama de caso de uso deve ter um assunto, caso de uso, atores e relacionamentos.

Principais componentes: ator, caso de uso, relacionamentos.

Como esses componentes já foram definidos, reforçaremos apenas os conceitos de relacionamento. Se você tiver alguma dúvida sobre os demais itens, releia as definições formais já apresentadas e reveja os exemplos.

Um relacionamento representa os itens relacionados a um caso de uso e/ou um ator. Figura também que tipo de relação há entre dois itens. Sempre que tivermos um relacionamento entre dois casos de uso, estes devem ser obrigatoriamente um include, um extend ou uma generalização.

Vamos tratar agora dos relacionamentos include e extend.

O include é um relacionamento de dependência que, como o próprio nome diz, é de inclusão, isto é, indica que o caso de uso de onde parte o relacionamento sempre inclui/executa o comportamento do outro caso de uso, que é apontado pela seta.

O extend é um relacionamento de dependência que poderá ter o comportamento da classe apontada extendido pela classe que aponta, como você pode observar na figura 111.

Como podemos ver na figura 111, a implementação do caso de uso Validar login implica necessariamente na execução dos casos de uso Validar usuário digitado e Validar senha.

Figura 111
Exemplo da utilização de relacionamentos include/extend.

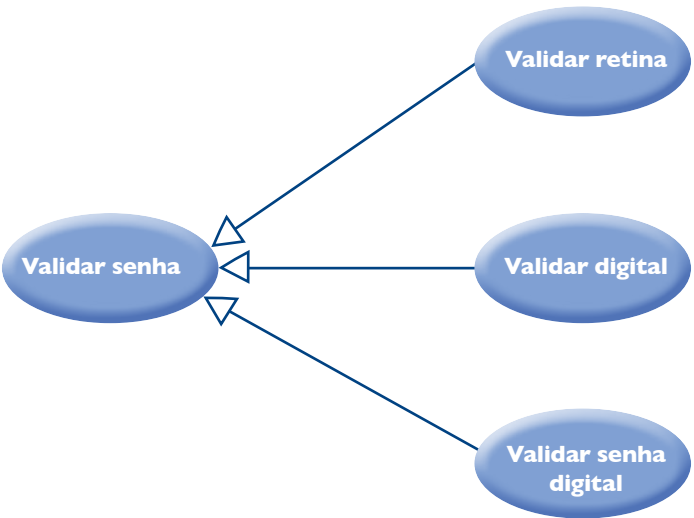
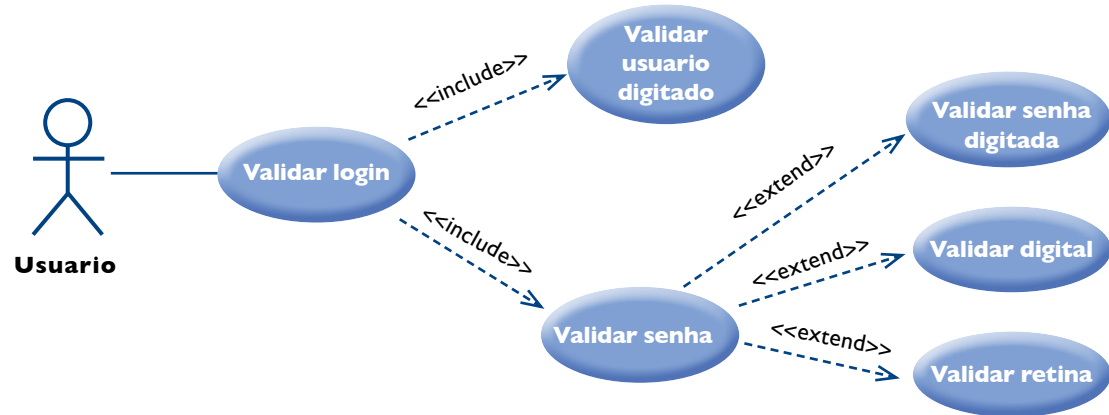



Figura 112
Relação de Generalização Especialização.

Já o caso de uso validar senha pode implicar em um dos três casos de uso. Dependendo do tipo de ação efetuada pelo usuário, será extendido o comportamento do caso de uso Validar senha digitada, validar digital ou validar retina.

Uma característica interessante que a UML nos oferece é ser extremamente flexível, possibilitando que utilizemos os blocos de construção de forma diferente, conforme a visão que queremos analisar no momento.

Voltemos ao exemplo da figura 111. O caso de uso validar senha, com o foco que descrevemos no diagrama, nos mostra uma relação extendida com os casos de uso Validar senha digitada, Validar digital ou Validar retina.

Também podemos escrever a relação entre esses casos de uso como se fosse uma relação de Generalização/Especialização. Assim, dependendo do enfoque que quisermos analisar, podemos combinar os elementos UML. Veja na figura 112 como ficaria com essa abordagem o fragmento do diagrama.

Podemos ter diagramas de caso de uso demonstrando diferentes níveis de abstração do sistema. Por exemplo, é possível ter um diagrama que represente o sistema como um todo, para poder analisar suas principais funcionalidades, como elas se agrupam, seus limites e a relação dos atores em cada um dos casos macro de uso.

Vamos retomar o estudo de caso da padaria do senhor João. Veja na figura 113 como ficaria um diagrama de caso de uso que representa as funcionalidades gerais a serem implementadas pelo sistema pedido.

Observe no diagrama que o retângulo no qual os casos de uso estão inseridos representa o sistema que estamos estudando. Seus limites são claros.

Os casos de uso representam as funções macro que devem ser implementadas pelo sistema, de acordo com as solicitações do senhor João.

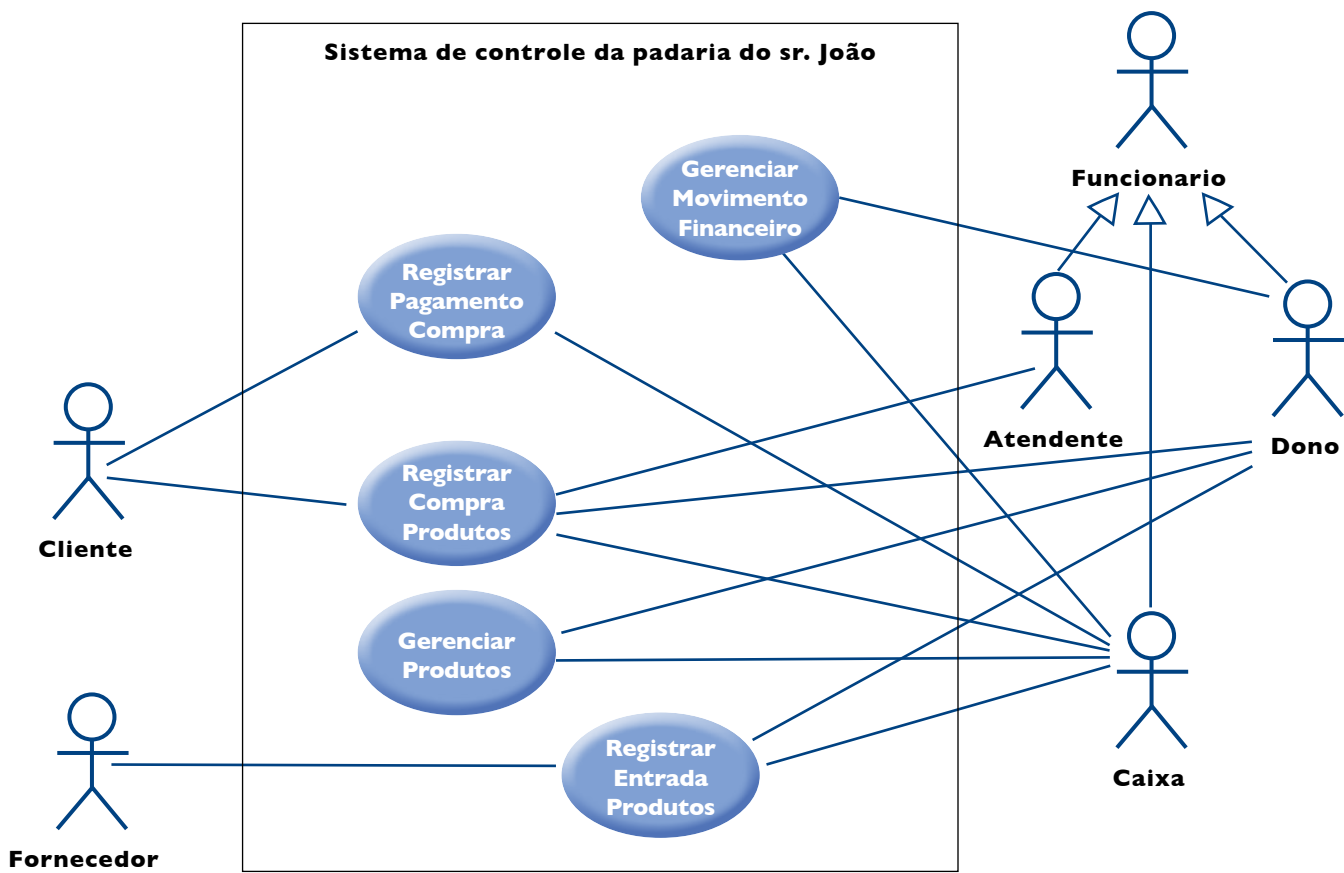



Figura 113
Diagrama de casos de uso do sistema de controle da padaria do senhor João.

Note que os atores podem ser representados isoladamente, como é o caso dos atores Cliente e Fornecedor, ou já indicando uma relação de generalização/especialização, como no caso da relação entre Funcionario, Caixa, Atendente e Dono.

Que outras informações podemos extrair desse diagrama?

Podemos ver que tanto o Caixa quanto o Atendente e o Dono (senhor João) podem registrar um produto vendido, mas apenas o Caixa e o Dono estão aptos a registrar entrada de produtos do Fornecedor.

Veja quantas informações importantes esse diagrama nos oferece. Mas ficou claro para você como cada um desses casos de uso devem ser implementados? Provavelmente não, porque ainda estamos com uma visão macro do problema e precisamos "descer" para outro nível de abstração.

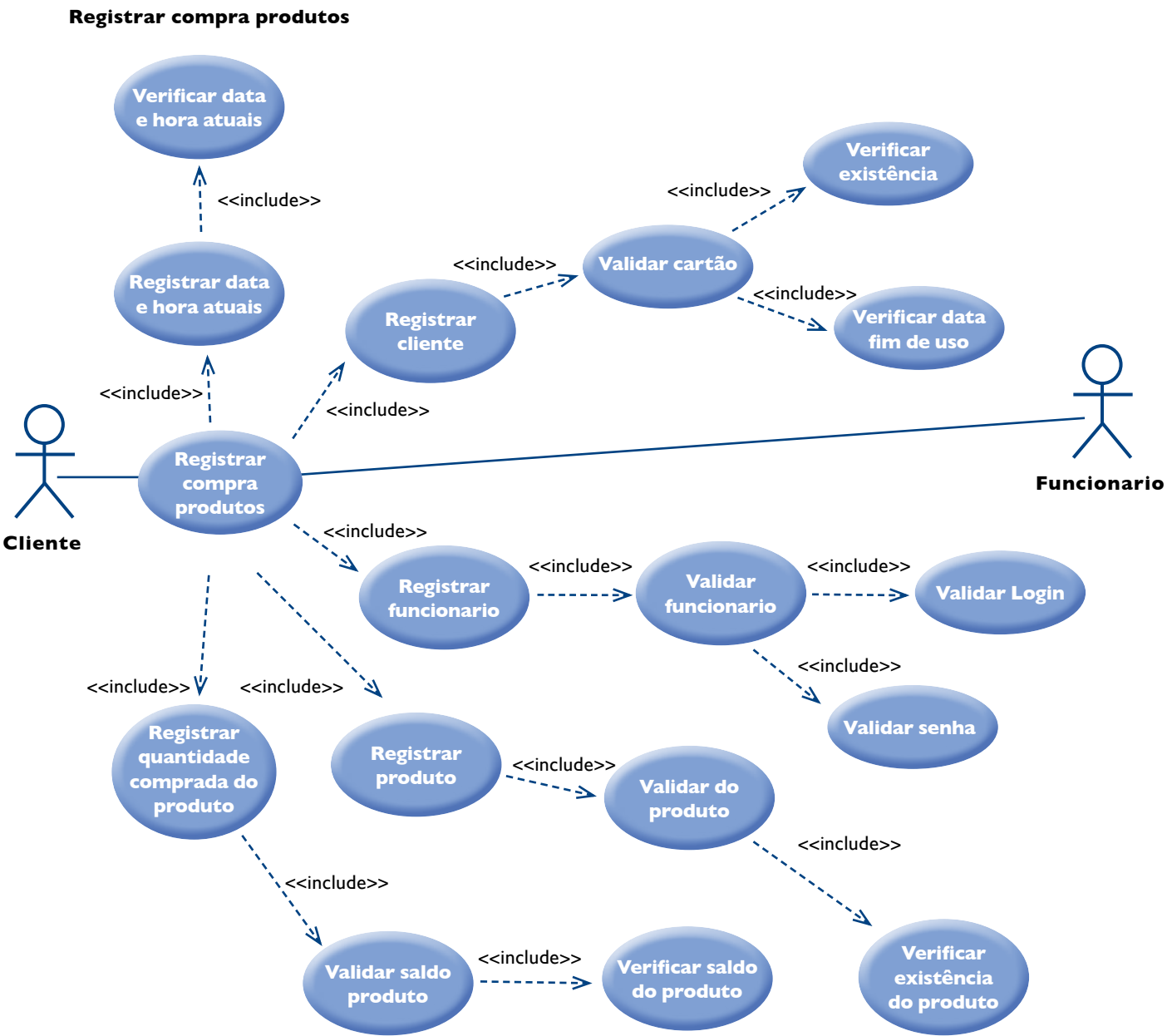
É nesse momento que devemos ter bom senso para perceber até onde devemos ir na construção de níveis mais baixos de abstração, de modo que não empreguemos tempo demais criando diagramas desnecessários para situações que já estão bastante claras.

O próximo passo será criar um diagrama para cada caso de uso listado, mostrando mais detalhadamente seu funcionamento.

Vamos analisar agora o caso de uso Registrar compra produtos (figura 114).

Nesse diagrama foi feito um detalhamento do caso de uso Registrar compra produtos. Analisando o diagrama pode-se verificar que, para registrar uma compra, é preciso registrar data e hora atuais, registrar o cliente, por intermédio do número do cartão, registrar o funcionário, confirmando seu login e senha, registrar o produto, confirmando sua existência, e registrar a quantidade vendida, confirmando se há saldo suficiente para fazer a venda.

Observe que esse diagrama já possui muitas informações, talvez até mais do

Figura 114
Caso de uso Registrar compra produtos.

que deveria. Se em algum momento você achar que seu diagrama está muito complexo, diminua nele o número de casos de uso.

Neste exemplo poderíamos nos limitar aos casos de uso Registrar data e hora atuais, Registrar cliente, Registrar produto e Registrar funcionário e, então, em um novo nível, mostrar os casos de uso que implementam cada um deles. Mas isso não seria necessário, pois os casos de uso estão claros e o diagrama não está muito poluído, isto é, não possui excesso de componentes que possa atrapalhar e compreensão.

Como já vimos, cada um dos casos de uso devem ser acompanhados por um descritivo. Existem várias propostas sobre como deve ser este descritivo. A que apresentaremos aqui propõe divisões, nome, atores envolvidos, pré-condições, fluxo básico e extensões.

Nome: é o nome do caso de uso, geralmente iniciando por um verbo.

Atores envolvidos: listar os atores e os papéis executados por eles no atual caso de uso.

Pré-condições: descrever o que é necessário para que se inicie a execução do caso de uso.

Fluxo básico: os passos a serem seguidos para a finalização correta do caso de uso.

Extensões: outras possibilidades de execução.

Observações: qualquer informação necessária para ajudar a compreender o funcionamento do caso de uso.

Agora vamos a um exemplo do caso de uso Registrar compra produtos. Veja:

Nome: Registrar compra produtos.

Atores envolvidos:

- Funcionário: ator que fará o registro da compra no sistema.
- Cliente: o cliente o qual a compra será registrada.

Pré-condições:

- Produto deve estar cadastrado.
- Cartão deve ser válido.
- Funcionário deve ter acesso ao sistema.
- Funcionário deve saber seu login e senha.
- A data e hora do sistema estão corretos.

Fluxo básico:

- Cliente chega para o funcionário com o produto e o cartão para registrar a compra.

- Funcionário recebe o cartão e o produto a ser registrado.
- Funcionário faz o login no sistema.
- Funcionário escolhe a opção de registro de compras no sistema.
- Funcionário passa o leitor de código de barras no cartão do cliente.
- Funcionário passa o leitor de código de barras no produto.
- Funcionário digita a quantidade comprada do produto.
- Funcionário confere dados cadastrados digitados.
- Funcionário confirma entrada da compra.

Extensões:

- A entrada do produto e da quantidade pode ser feita pela balança quando se tratar de produtos pesados na padaria.

Observações:

Não há.

4.3.2. Diagrama de classes

Um diagrama de classes mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos. **Os diagramas de classes** abrangem a visão estática de projeto de um sistema. Expõem a coleção de elementos declarativos (estáticos) (BOOCH, RUMBAUGH e JACOBSON, 2005, *UML – Guia do usuário.*)

O Diagrama de classe não deve faltar em projetos orientados a objetos. Devemos prestar muita atenção ao criar um diagrama de classes, que será a base da nossa solução.

Principais componentes: classes, interfaces, relacionamentos.

O diagrama de classes fornece uma visão estática do modelo a ser criado. Como as classes são um dos componentes mais importantes da orientação a objetos, esse diagrama deve constar de todo projeto orientado a objetos.

Identificar uma classe não é tarefa das mais simples, mas o caminho é procurar itens que têm as mesmas informações e comportamentos. Nem sempre uma classe tem atributos e métodos. Pode ter apenas métodos ou apenas atributos.

Tente fazer uma lista do que você identificou como classes. Acrescente os atores, que geralmente são também classes de seu sistema.

Analise as candidatas a classes e tente achar atributos para elas e, se possível, alguma funcionalidade.

Coloque as classes em um diagrama e comece a analisar as relações entre elas, de acordo com os tipos de relacionamentos que você estudou.

Lembre-se de que todos esses componentes foram definidos anteriormente. Se tiver dúvidas, volte a ler as definições sobre o tema.

Vamos agora analisar uma parte do diagrama de classes da padaria do senhor João. Essa parte foi montada com ênfase nos casos de uso Registrar compra e pagar compra. Por isso, serão mostrados somente os métodos envolvidos na

execução desses casos de uso (veja figura 115). As classes que não participam deles são apresentadas apenas como seus atributos.

Podemos identificar vários elementos da teoria de orientação a objetos nessa parte do diagrama. Vemos aí exemplos de generalização/especialização entre as classes Funcionario, Dono, Atendente e Caixa. E também de composição

Figura 115
Diagrama de classes.

Fornece
- dataEntrada : date
- nroNF : int
- valorTotal : double
- lancadoPor : Funcionario
- dataLancamento : date
- horaLancamento : time
- fornecedor : Fornecedor

ItemFornece
- produto : Produto
- quantidade : double
- valorUnitario : double

Fornecedor
- codigo : int
- nome : String
- cnpj : String

Produto
- codigo : int
- descricao : String
- quantidadeEstoque : double
- vlUnitarioVenda : double
- estoqueMinimo : double
- situacao : int
+ verificaProduto9produto : Produto, quantidade : double) : double

Funcionario
- cpf : String
- nome : String
- nroCarteiraProfissional : int
- dataAdmissao : date
- dataDemissao : date
- senha : String
+ verificaFuncionario(cpf : String, senha : String) : boolean

Cliente
- nroCartão : int
- datainicioUso : date
- dataFimUso : date
+ verificaCartao(nroCaetao : int) : boolean

ItemCompra

ItemCompra
- quantidade : double
- dataRegistro : date
- horaRegistro : time
- atendidoPor : Funcionario
- cliente : Cliente
+ atualizaTotalCompra(valorTotalItem : double) : boolean

Atendente
- salarioHora : double

Caixa
- salarioMensal : double

Compra
- dataCompra : date
- valorTotal : double
- terminada : boolean
- recebidoPor : Funcionario

Dono
- proLabore : double

entre Compra/ItemCompra e Fornece/ItemFornece, além de uma classe de associação ItemCompra.

Podemos incluir a multiplicidade nos relacionamentos, se quisermos analisar esse requisito, para, por exemplo, projetarmos o banco de dados relativo a solução.

O diagrama de classes oferece inúmeras visões de nosso projeto, que vão desde a visão da relação entre as classes até a das abstrações utilizadas. E pode, até mesmo, ajudar na criação do banco de dados vinculado à solução.

Dependendo do foco da análise, podemos exibir os detalhes desse diagrama de forma diferente – os estereótipos das classes, seus atributos, métodos, responsabilidades ou apenas uma dessas características.

4.3.3. Diagrama de sequência

É um diagrama de interação que dá ênfase à ordenação temporal de mensagens. (BOOCH, RUMBAUGH e JACOBSON, 2005).

O diagrama de sequência permite a análise e definição da troca de mensagens entre os objetos e atores da solução e é muito utilizado para definição de parâmetros e classes dos métodos a serem criados.

Devemos estabelecer um diagrama de sequência para cada caso de uso cujo funcionamento tenhamos dificuldade de entender ou tenhamos dúvidas a respeito de como implementá-lo.

Principais componentes: atores, classes, objetos, mensagens e focos de controle.

Foco de controle é um componente do diagrama de sequência que permite a representação de comandos de decisão, de loop e de opção. A ideia é que todos os fluxos que se encontrarem dentro do foco de controle sejam executados quando ou enquanto a condição exibida for verdadeira. Veja a figura 116.

As mensagens representam a comunicação entre os objetos e atores do diagrama. São simbolizadas graficamente por setas. Existem quatro tipos de mensa-

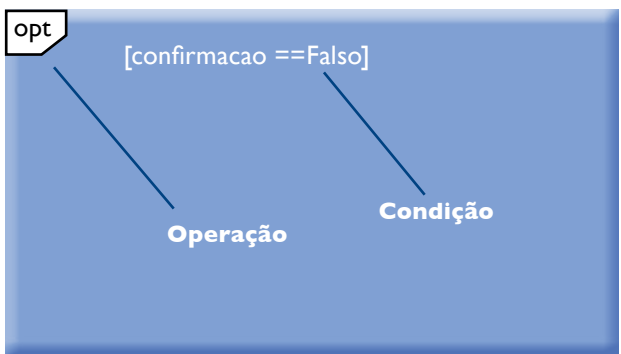
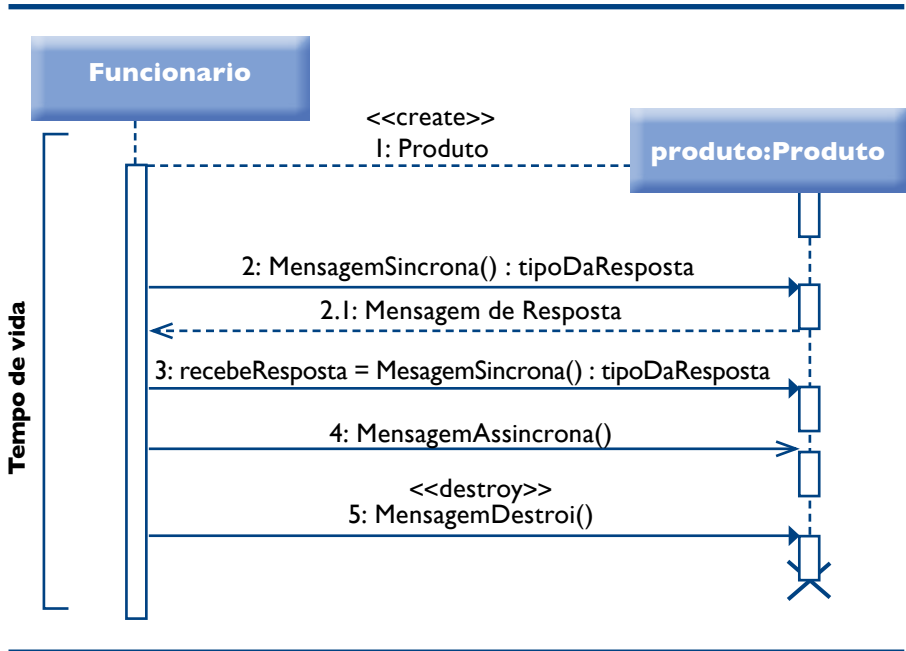



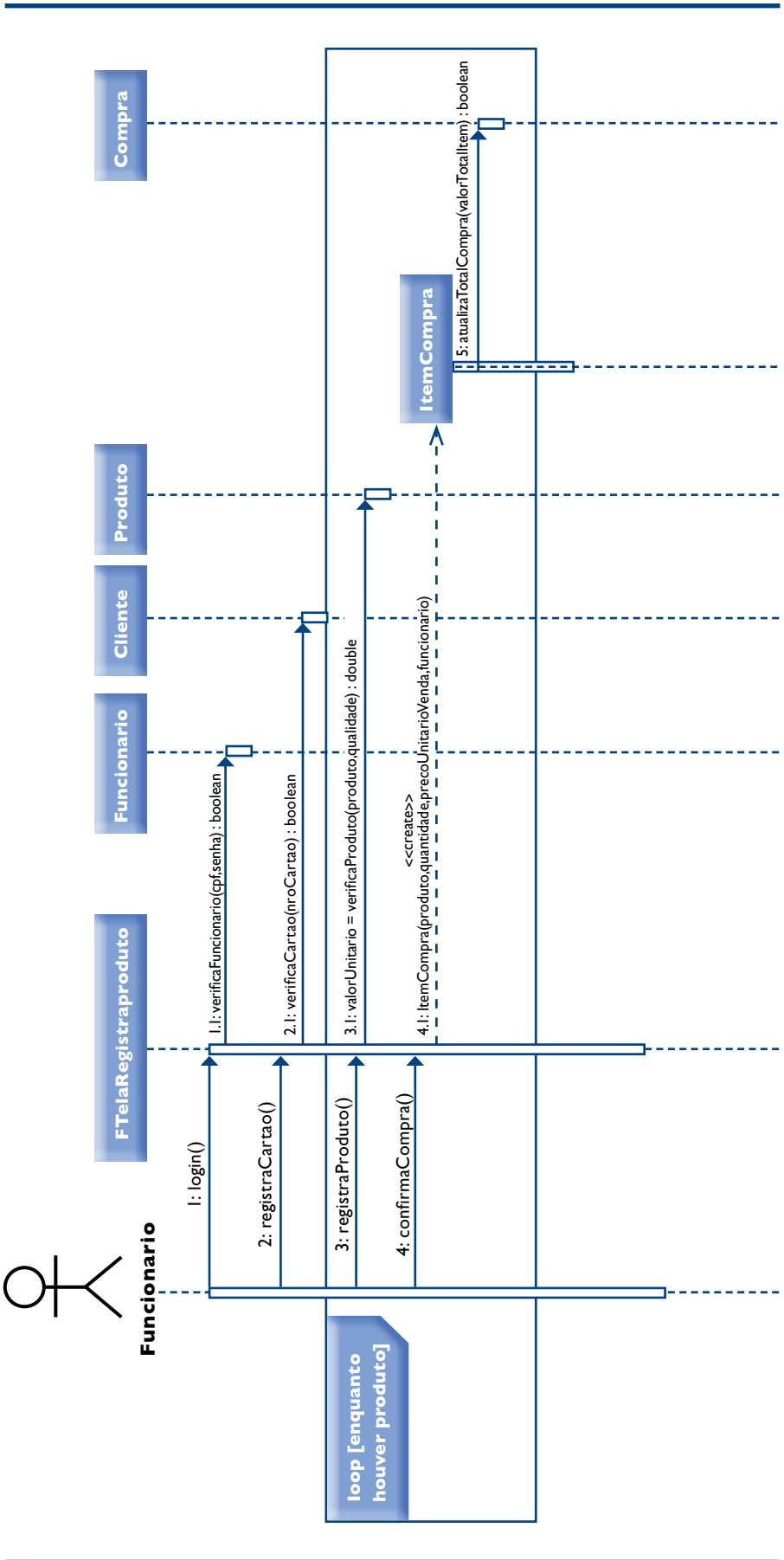
Figura 116
Representação de um foco de controle.

Figura 117
Troca de mensagens e tempo de vida em um diagrama de sequência.

gens definidas. Mensagens síncronas são as que aguardam o retorno do fim de seu processamento para continuar a execução; mensagens assíncronas são aquelas que o remetente continua executando sem aguardar resposta; e há também as mensagens de create e destroy, que representam as chamadas dos métodos construtor e destrutor das classes.

A figura 117 mostra os quatro tipos de mensagem em um diagrama que representa a troca de mensagens entre Funcionario e Produto.

A primeira mensagem é a de criação de um objeto da classe Produto. Em seguida, vemos uma mensagem síncrona indicando que o remetente aguarda que o receptor processe a mensagem antes de continuar seu procedimento.
A mensagem número 1 é a de criação de um objeto da classe Produto, aquela que chama o método construtor da classe Produto.
A número 2 é uma mensagem síncrona, isto é, que aguarda o retorno de uma informação para continuar sua execução normal. Veja que o retorno pode ser por meio do final da execução do próprio método, da definição de um "tipoDaResposta, diferente de void ou ainda de uma mensagem de resposta, como a indicada na figura com o número 2.1.
A mensagem número 3 também é síncrona, mas nomeia o retorno pela execução do método. Isso permite melhor visualização da execução, principalmente quando se trata de retorno que define seu fluxo.
A quarta é uma mensagem assíncrona. Ou seja, é enviada, mas o emissor não aguarda o retorno da mensagem para continuar seu fluxo de execução.
A número 5 é uma mensagem que destrói o objeto criado. No exemplo, destrói o objeto produto criado e demonstra a chamada do destrutor da classe.
Podemos ver também, na figura 117, que a numeração das mensagens possibilita a compreensão da ordem em que são executadas.

Vamos agora analisar o diagrama de sequência que trata do caso de uso Registrar compra produtos (figura 118).

Figura 118
Diagrama de sequência do caso de uso Registrar compra produtos.

Analisando o diagrama, vemos que no início o funcionário faz o login no sistema e informa o número do cartão do cliente. Quando os dados do produto comprado foram digitados, as mensagens foram inseridas em um foco de controle que sugere a implementação de um loop, o qual, por sua vez, indica que aquela troca de mensagens ocorrerá enquanto houver produtos a serem lançados para o cliente.

Veja que agora sabemos exatamente quais métodos as classes envolvidas nesse caso de uso devem implementar.

Volte agora ao diagrama de classes. Observe que os métodos criados naquele diagrama saíram deste.

Note que a cada compra confirmada é criado um novo objeto itemCompra.

Neste exemplo podemos ver a definição da sequência de execução das classes como também da interface gráfica (GUI – Graphical User Interface), representada pela classe TelaRegistraProduto.

Volte agora ao exemplo do diagrama de classes e reveja os métodos definidos para as classes envolvidas no diagrama. Você perceberá que tais métodos foram definidos a partir desse diagrama de sequência.

4.3.4. Diagrama de comunicação

É um diagrama de interação que dá ênfase à organização estrutural de objetos que enviam e recebem mensagens; um diagrama que mostra as interações organizadas ao redor de instâncias e os vínculos entre elas. (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: objetos, mensagens.

O diagrama de comunicação mostra a relação entre os objetos, analisando a troca de mensagens entre eles. Mas não se preocupa necessariamente com a ordem em que elas ocorrem, e sim com quais objetos as mensagens são trocadas e quais são as mensagens.

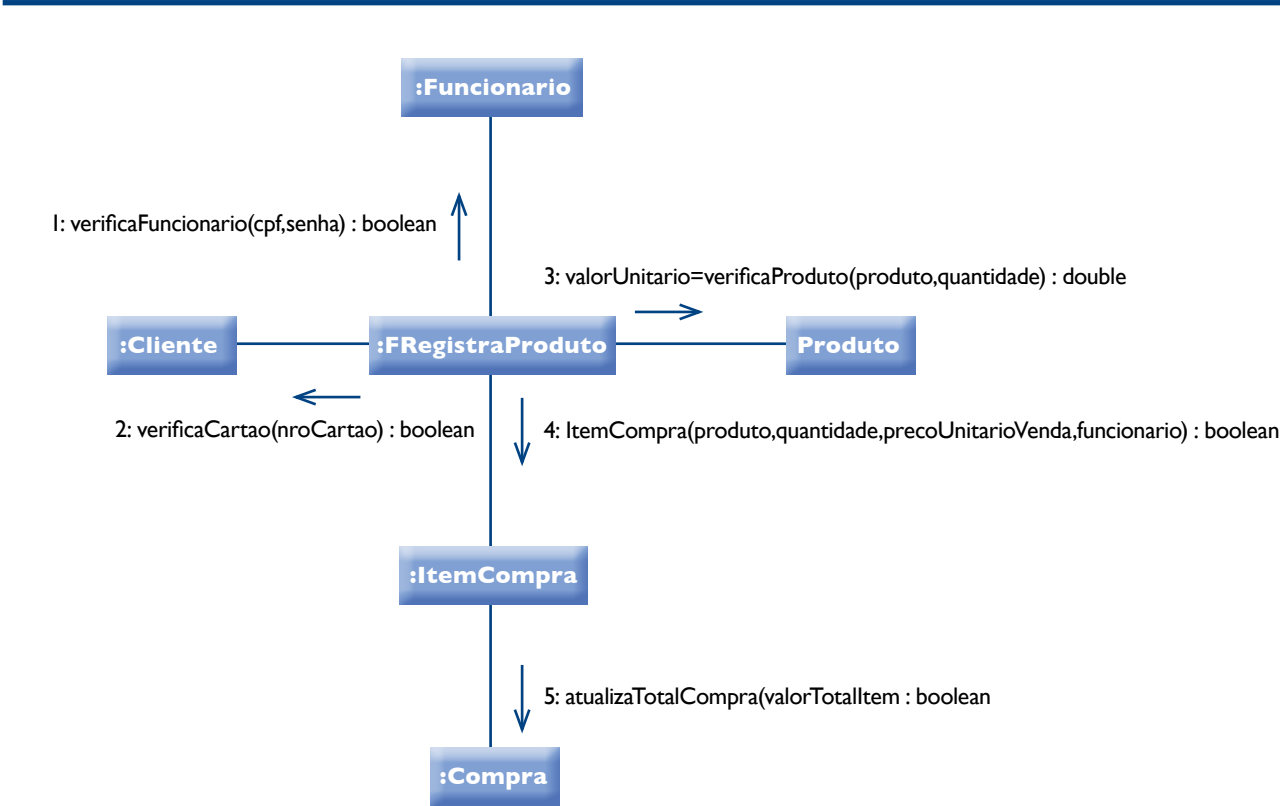
Vamos analisar um diagrama de comunicação tomando por base o caso de uso Registrar compra produtos (figura 119).

Veja que ele traz, basicamente, as mesmas informações do diagrama de sequência, mas em ordem diferente, dando ênfase às mensagens trocadas pelos objetos, mas não necessariamente quando essas mensagens são trocadas.

Em muitos projetos, usa-se ou o diagrama de sequência ou o diagrama de comunicação, mas também pode-se criar ambos para análise de alguma situação particular.

4.3.5. Diagrama de atividades

É um diagrama que mostra o fluxo de controle e dados de uma atividade para outra; os diagramas de atividades abrangem a visão dinâmica do sistema. (BOOCH, RUMBAUGH e JACOBSON, 2005).

Figura 119
Diagrama de comunicação do caso de uso Registrar compra de produtos.

Por modelar aspectos dinâmicos do sistema, esse diagrama permite análise e documentação da sequência de atividades envolvidas em um caso de uso ou mesmo em um fluxo de trabalho. Possibilita a visão dos procedimentos efetuados para execução de um caso de uso, por exemplo, dando acesso a documentação dos procedimentos, tanto do sistema quanto das atividades extrassistema.

Todo diagrama de atividades deve possuir um início, marcado por um círculo preenchido, e um fim, representado por um círculo preenchido, porém com um aro branco na extremidade.

Existem também as ações que são representadas por um retângulo de bordas arredondadas, tendo em seu interior o nome da ação executada.

A execução de uma ação pode ser condicional a alguma ocorrência. Esses desvios condicionais são representados por um losango com as setas partindo para as ações a serem executadas, seja a condição satisfeita ou não.

Deve-se criar diagrama de atividades para os casos de uso cujo funcionamento não está claro ou para documentar os procedimentos a serem seguidos para sua execução.

Principais componentes: atividades, decisões, fluxos.

Veja na figura 120 o exemplo do diagrama de atividades gerado pelo caso de uso Registrar compra produtos.

Observe que o diagrama de atividades apresenta uma forma simples de documentar as ações executadas em cada caso de uso. É, asssim, uma importante ferramenta de documentação do software que está sendo produzido. Note também que você deve dividir o diagrama com linhas verticais para identificar quem deve fazer a ação.

4.3.6. Diagrama de pacotes

O diagrama de pacotes mostra a decomposição do próprio modelo em unidades organizacionais e suas dependências (BOOCH, RUMBAUGH e JACOBSON, 2005).

É um diagrama estrutural que permite uma visão da organização interna da aplicação que está sendo projetada.

Principais componentes: pacotes.

Como vimos anteriormente, dentro de um pacote podemos inserir quaisquer componentes da UML. Assim, podemos criar pacotes para estruturar nossa aplicação, usando sua modularização para organizá-la e facilitar sua compreeensão.

Veja o exemplo da figura 121.

Neste exemplo, usamos pacotes para organizar o projeto, separando as classes de projeto das de interface com o usuário (telas), também conhecidas como GUI (Graphical User Interface).

Como podemos colocar em pacotes todos os elementos da UML, devemos utilizá-los para organizar e modular nossos projetos, deixando-os mais claros e fáceis de compreender e manter.

4.3.7. Diagrama de gráficos de estados

Os diagramas de máquinas de estados abrangem a visão dinâmica de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: estado, evento.

Esse diagrama mostra os estados que podem ser assumidos por um objeto em seu ciclo de vida. Geralmente o utilizamos para entender como tais mudanças acontecem de modo a podermos definir as trocas de mensagens e os métodos que as controlam.

O início da transição é representado por um círculo preenchido e o final, por um círculo preenchido, porém com um aro pintado de branco.

Utilizaremos como exemplo a classe Produto do estudo de caso da padaria do senhor João, que pode assumir três estados diferentes: 1 Ativo, 2 Ponto de encomenda e 3 Em falta. Esses valores são aplicados ao atributo situação quando a execução do caso de uso Registrar pagamento compra ou do caso de uso Registrar entrada de produtos (veja figura 122).

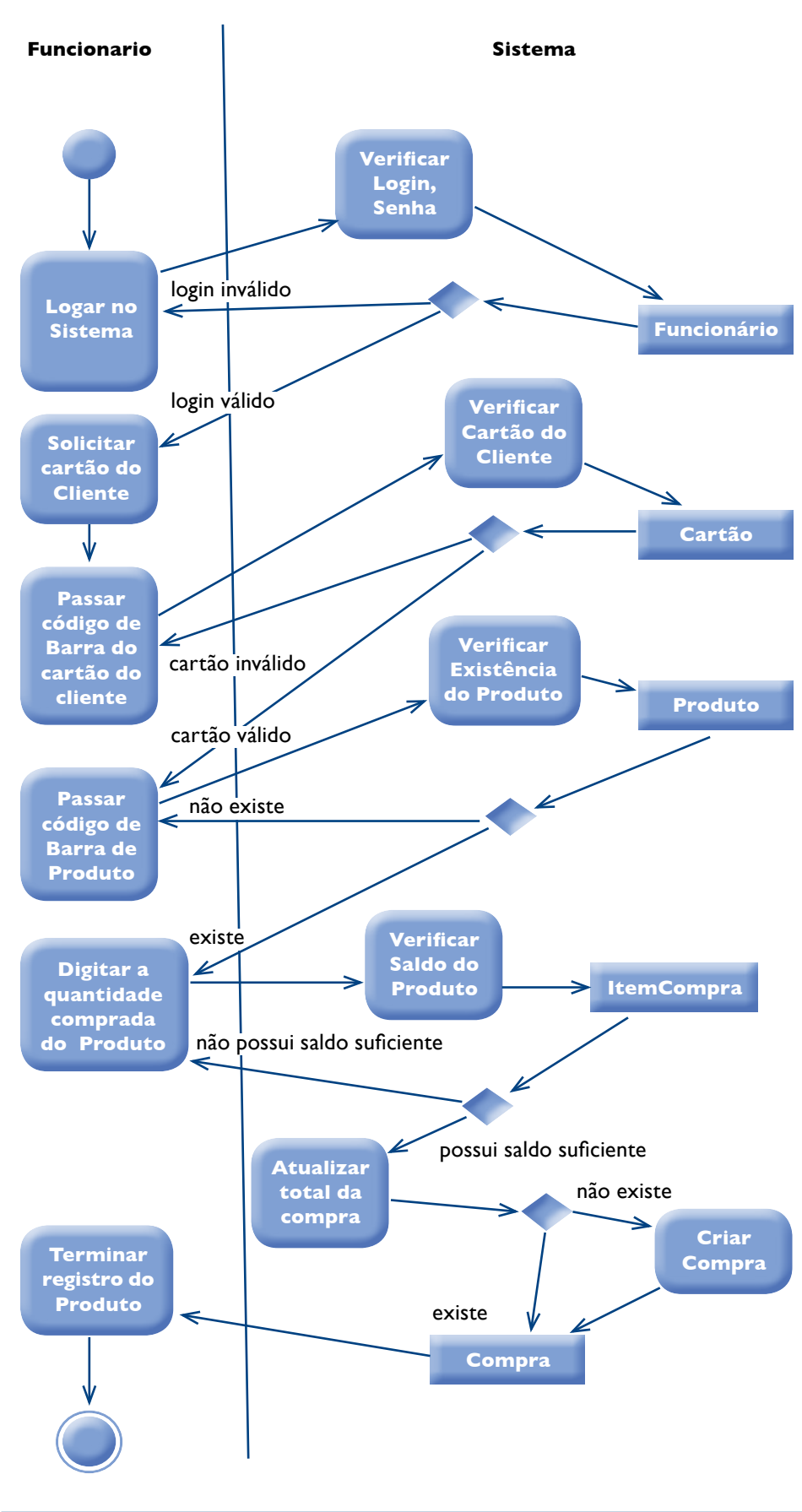



Figura 120
Diagrama de atividades do caso de uso Registrar compra produtos.

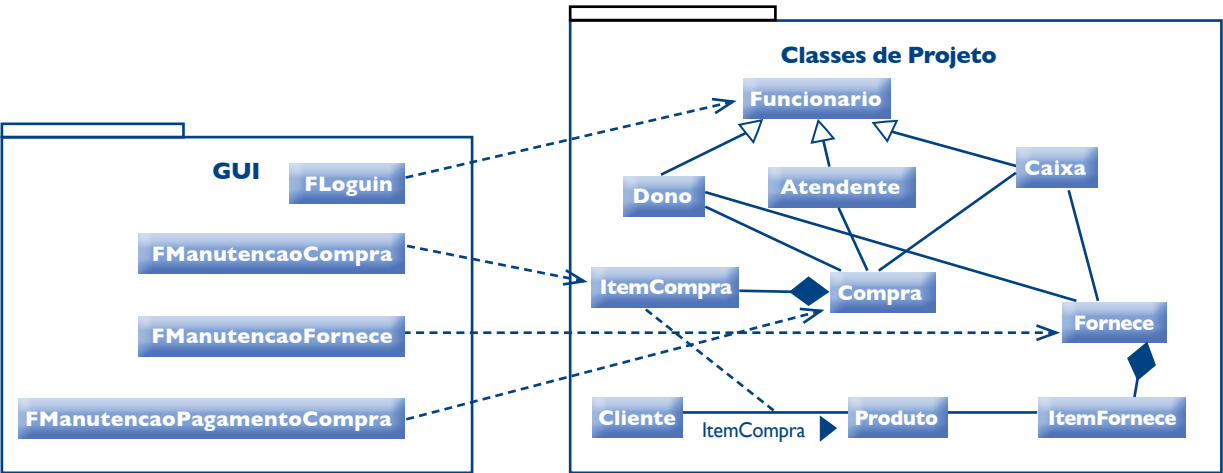



Figura 121
Diagrama de pacotes.

Analisando o diagrama podemos ver que a partir de um estado ativo, o produto poderá passar a ponto de encomenda ou em falta, dependendo das suas saídas e considerando-se que a regra para um produto chegar à condição de ponto de encomenda é seu saldo ser menor ou igual à indicada nesse campo.

Notamos também que só os métodos pagarCompra e receberProduto alteram o estado do produto e que a baixa do estoque só é efetuada quando se realiza o pagamento da compra.

Como pudemos observar, esse diagrama nos ajuda a entender e a definir melhor o funcionamento de nosso sistema quando há mudanças de estado dos objetos.

4.3.8. Diagrama de objetos

Mostra um conjunto de objetos e seus relacionamentos em um ponto do tempo; os diagramas de objetos abrangem a visão estática de projeto ou visão estática de processo de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Principais componentes: objetos, relacionamentos.

Este diagrama nos dá uma visão de como ficarão os objetos em determinado momento da execução do sistema. É como se tirássemos uma fotografia do sistema em um momento para analisar os dados e os relacionamentos envolvidos, como você pode observar na figura 123.

Observe a notação desse diagrama: o objeto possui a mesma estrutura de uma classe, porém seu nome vem antes do nome da classe. func: Funcionario quer dizer objeto func da classe funcionário.

Podemos assim analisar as relações entre os objetos em um determinado ponto da execução do sistema.

4.3.9. Diagrama de componentes

Mostra a organização e as dependências existentes em um conjunto de componentes; os diagramas de componentes abrangem a visão estática de implementação de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

O diagrama de componentes é um diagrama estrutural que nos ajuda a analisar as partes do sistema que podem ser substituídas por outras que implementem as mesmas interfaces (de entrada e/ou de saída) sem alterar o seu funcionamento.

Principais componentes: componentes, interfaces, classes e relacionamentos.

Todo componente, geralmente, pode ser substituído por uma classe, que implementa suas interfaces. Por isso é bastante difícil separar um do outro. Mas costumamos utilizar o diagrama de componentes quando precisamos documentar um componente que pode ser substituído no sistema.

Um claro exemplo de uso de componente e, consequentemente, do diagrama de componentes no estudo de caso da padaria do senhor João, é a representação da balança que pesa os produtos comercializados na padaria.

Como a balança vai interagir com os componentes do software, alimentando o sistema com informações sobre o produto vendido, como peso e valor, o senhor João poderá substituir a balança apenas por outra que possibilite as mesmas interfaces, tanto de entrada quanto de saída. Vejamos na figura 124 como representamos essa situação num diagrama de componentes.

Podemos aproveitar e definir as interfaces de entrada e saída da balança e deixá-las documentadas nesse diagrama.

Figura 122
Diagrama de gráfico de estados.

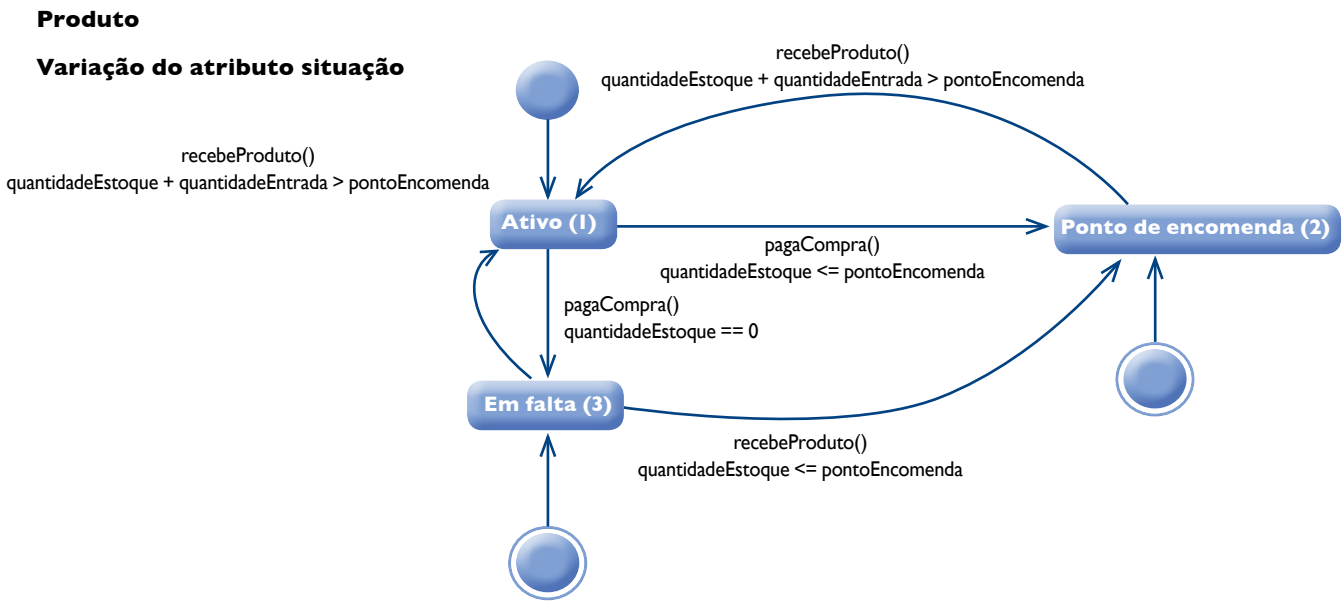
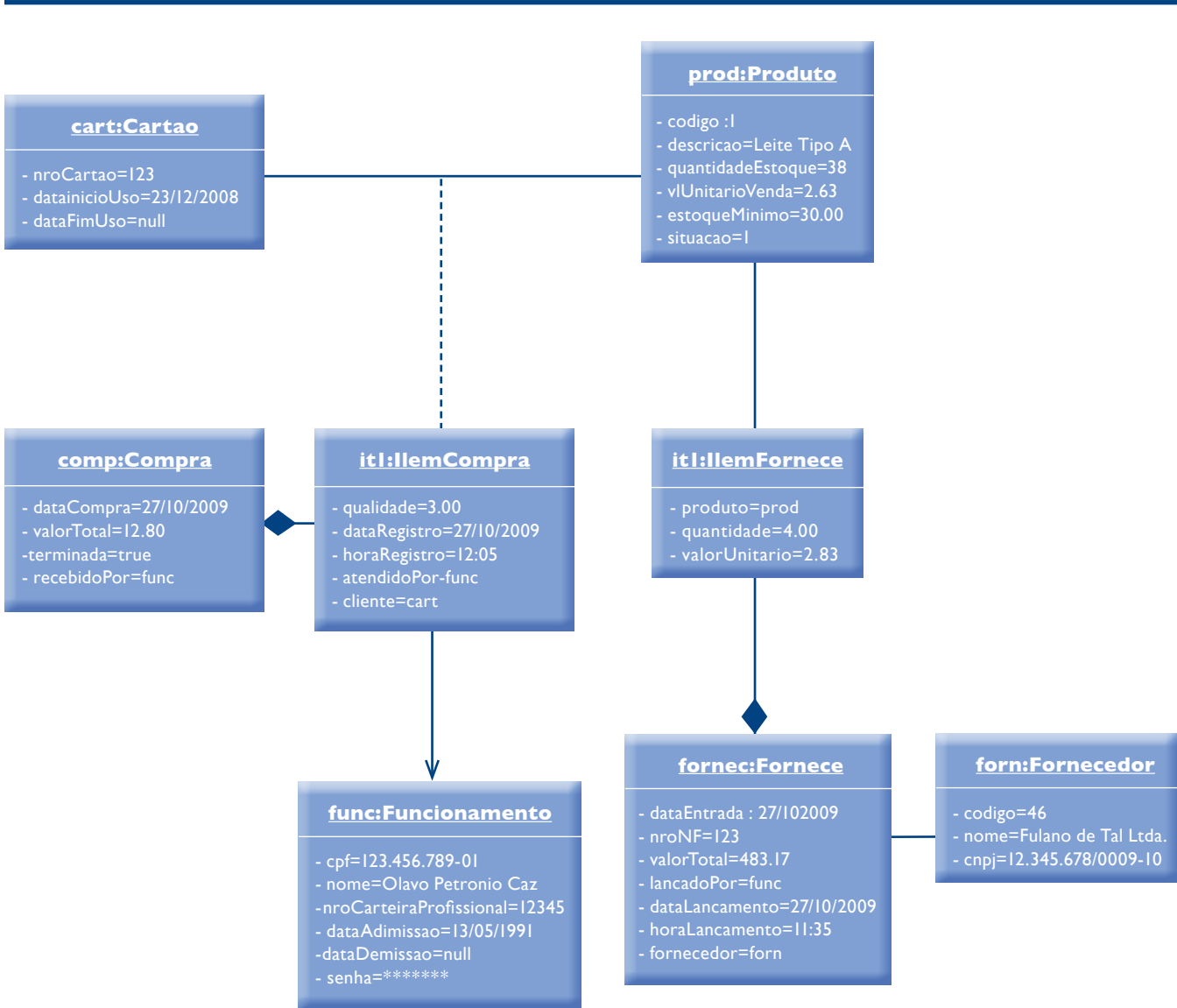



Figura 123
Diagrama de objetos.

4.3.10. Diagrama de implantação

Mostra a configuração dos nós de processamento em tempo de execução e os componentes envolvidos. Um diagrama de implantação abrange a visão estática de implantação de um sistema (BOOCH, RUMBAUGH e JACOBSON, 2005).

Trata-se de um diagrama estrutural que mostra como será criada a estrutura de software e hardware onde a solução será implementada. Podemos visualizar com esse diagrama toda a arquitetura da solução desde os servidores, sistemas operacionais, demais softwares e serviços requeridos, além dos protocolos de comunicação.

Principais componentes: nós, artefatos, relacionamentos.

Os nós podem representar dispositivos computacionais, como um computador ou um celular, ou um recurso de computação, como um sistema operacional,

um sistema gerenciador de banco de dados, um servidor de aplicação ou quaisquer outros softwares que integrem a estrutura da aplicação. Possuem um nome e podem receber um estereótipo. Um nó é representado por um cubo. Os nós executam os artefatos.

Artefatos são os elementos executados pelos nós, geralmente os programas da solução criada. Possuem um nome e podem possuir um estereótipo.

Os relacionamentos são utilizados para representar o tipo de ligação entre os componentes do diagrama. Podem possuir estereótipos.

Veja que, na figura 125, demonstramos em detalhes a arquitetura da solução proposta.

4.3.11. Diagrama de temporização

É um diagrama de interação, que mostra os tempos reais em diferentes objetos e papéis, em vez de sequências de mensagens relativas (BOOCH, RUMBAUGH e JACOBSON, 2005).

O diagrama de temporização, como o nome diz, tem seu foco principal no estudo do tempo gasto nas trocas de mensagens entre os componentes do sistema. É um diagrama de interação, pois auxilia o entendimento do processo de troca de mensagens entre os diversos componentes do sistema. Geralmente é utilizado em projetos de sistemas de tempo real, em que os tempos gastos nas trocas de mensagens e de estados na execução da tarefa são essenciais.

Esse é um diagrama de uso específico que não se utiliza em todos os projetos. Mas trata-se de um recurso que você deve conhecer caso precise analisar um sistema no qual o tempo seja um fator crucial.

Principais componentes: classes, linha de tempo, mensagens.

Criaremos um exemplo de diagrama de temporização para definir os tempos aceitáveis para as operações executadas pela balança da padaria do senhor João. A balança em questão vai interagir com o sistema, acessando os objetos da clas-

Figura 124
Diagrama de componentes.

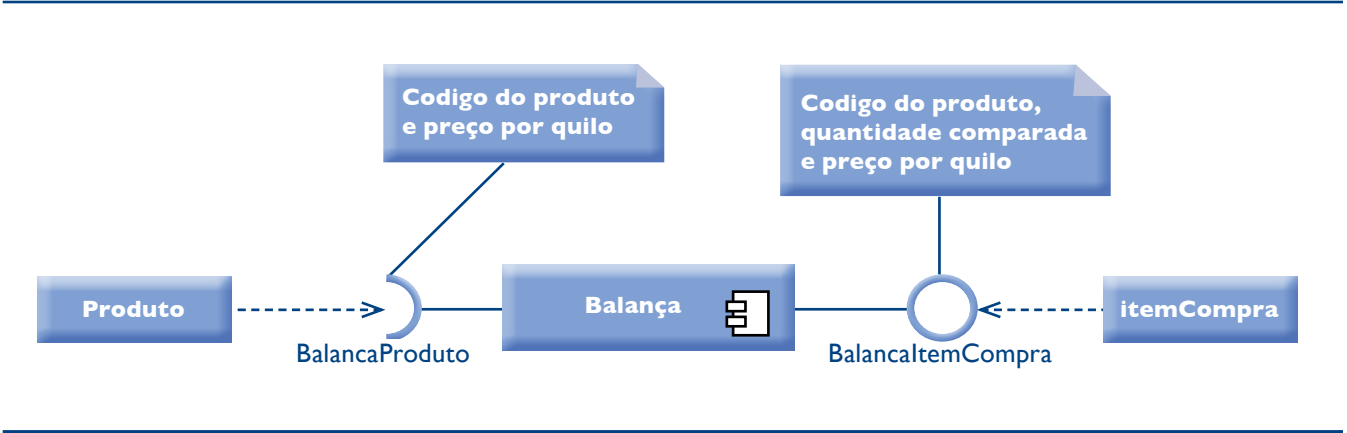
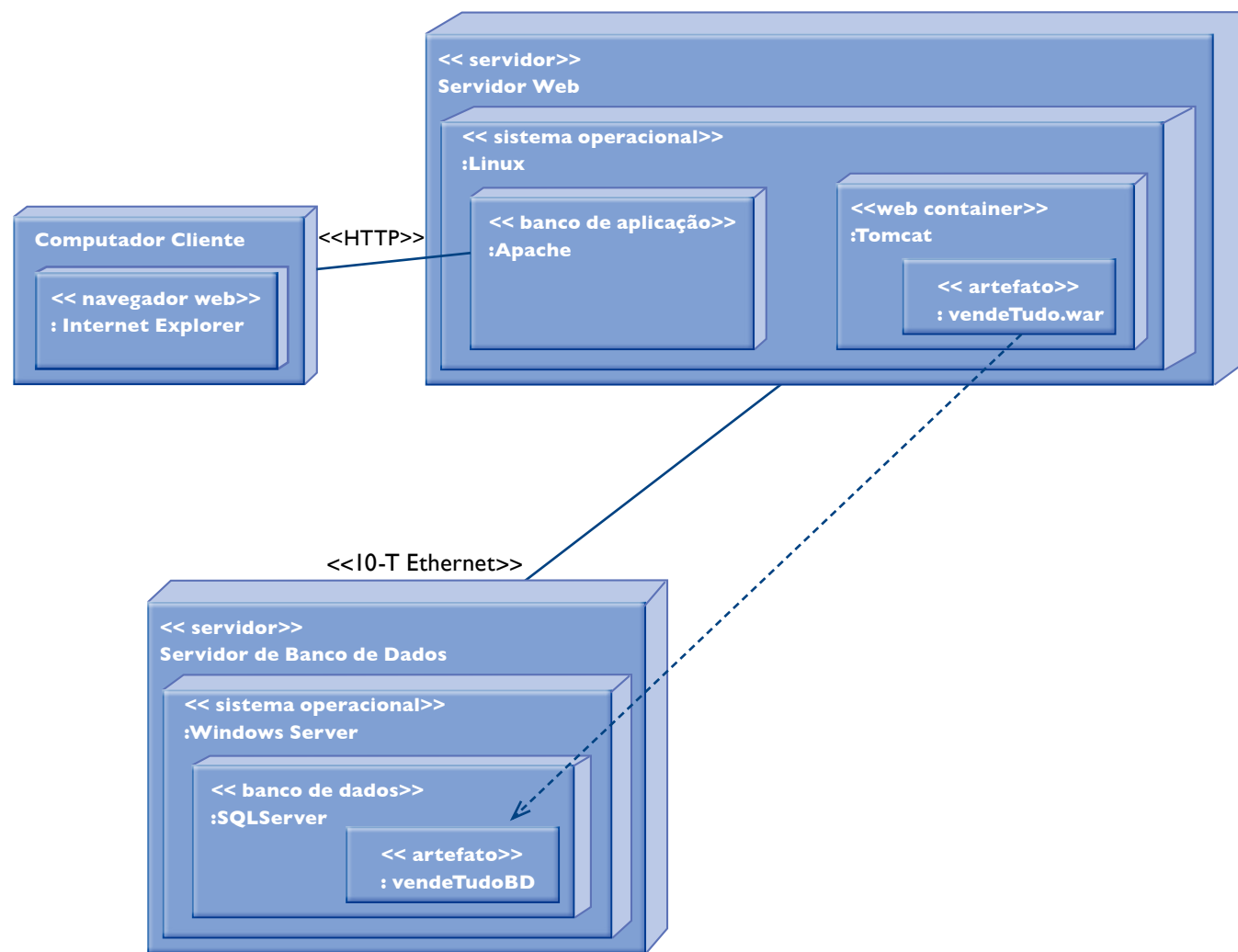



Figura 125
Exemplo de diagrama de implantação.

se produto, para pesquisar o preço por quilo. Depois da pesagem, calculará o valor do item e instanciará um objeto da classe ItemCompra. É exatamente essa situação que analisaremos no diagrama abaixo, avaliando os tempos aceitáveis para cada operação.

A balança possui um visor das informações digitadas/calculadas e um teclado numérico por meio do qual o atendente introduz os dados. Como apenas um atendente trabalha na pesagem a cada turno, ele faz o login na balança ao iniciar o trabalho e, assim, não terá de se identificar toda vez que precisar pesar algum produto.

Veja como representamos esse diagrama, na figura 126.

Podemos perceber que nosso foco na análise do tempo gasto é nas operações efetuadas pela balança, sem a intervenção do usuário.

As operações em foco são pesagem do produto, obtenção do preço por quilo,

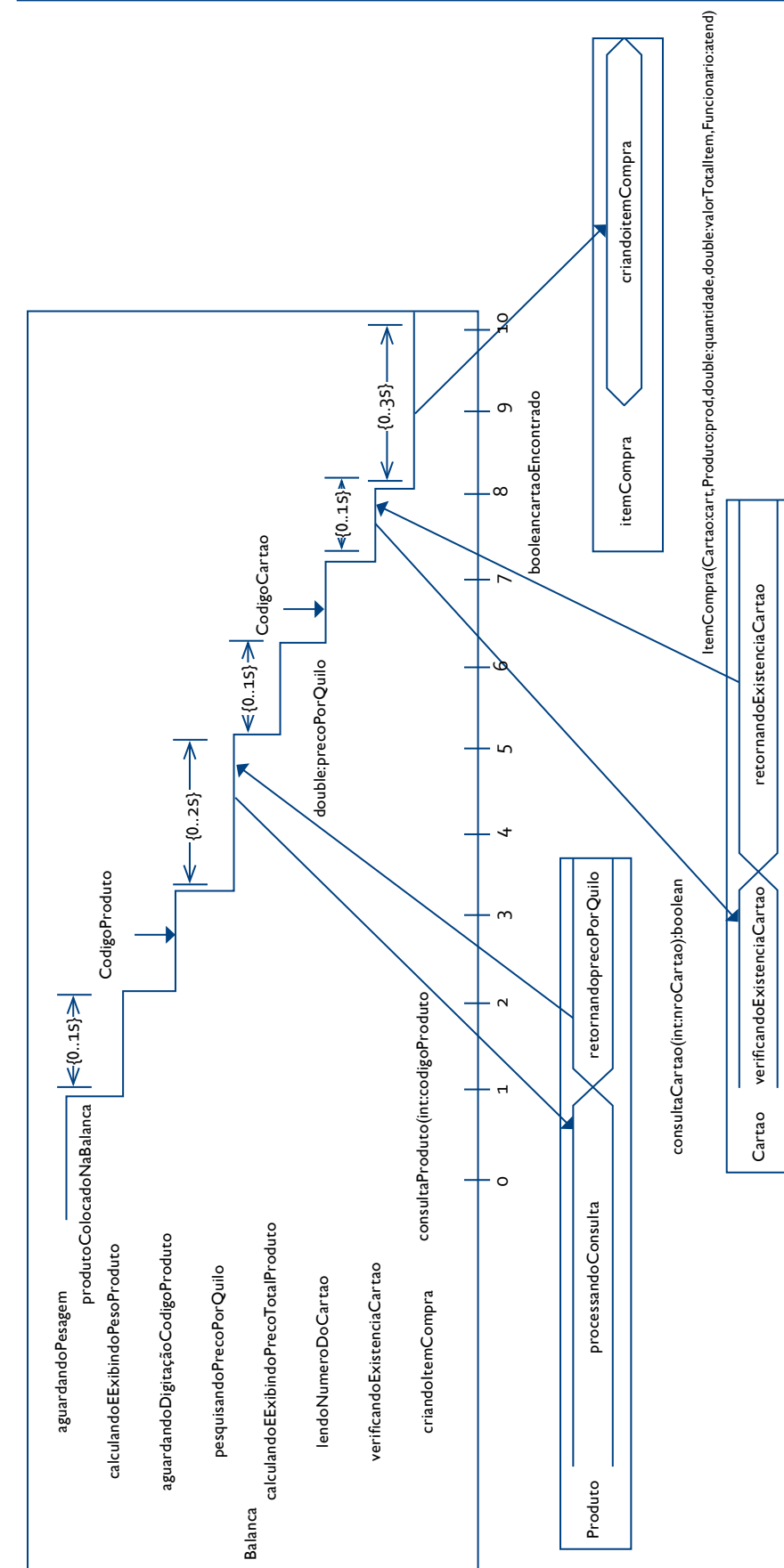



Figura 126
Diagrama de temporização.

cálculo do preço do item e geração do registro da compra (criação da instância da classe ItemCompra). São as únicas operações para as quais o diagrama demonstra um tempo máximo – as demais, que dependem da interação do atendente, têm os tempos estimados e apenas grafados na linha de tempo para podermos ter uma ideia do tempo total gasto com a operação.

Sabemos agora que requisitos de tempo básicos a balança deve realizar para que possa ser utilizada neste sistema.

4.3.12. Diagrama de estrutura composta

É utilizado para exibir as classes, interfaces e relacionamentos criados para implementar uma colaboração. Faz parte dos diagramas de interação.

Principais componentes: classes, relacionamentos e colaborações.

O diagrama de estrutura composta permite identificação e análise das classes e demais componentes que constituem uma colaboração.

Uma colaboração, como já vimos, é um agrupamento de classes, relacionamentos e interfaces que constituem uma unidade do sistema.

Vamos analisar o diagrama de estrutura composta (figura 127), que mostra as classes envolvidas na colaboração Receber produto.

Veja que no diagrama representado na figura 127, estamos analisando a colaboração Receber_Produto. Podemos verificar todas as classes envolvidas com a implementação dessa colaboração e as relações entre elas. Isso nos permite analisar cada colaboração em um nível mais baixo da abstração, isto é, de forma mais clara e detalhada. Podemos detalhar mais, se for necessário, incluindo ainda as classes de interação com o usuário e as mensagens.

Figura 127
Diagrama de estrutura composta.

Esse diagrama não é muito utilizado, mas pode ser útil para a compreensão da forma de implantar uma colaboração.

4.3.13. Diagrama de visão geral de interação

Tem por objetivo analisar as sequências necessárias para executar determinada funcionalidade do sistema que estamos projetando. Tal funcionalidade pode ser uma operação envolvendo vários casos de uso. Como seu nome diz, este diagrama faz parte dos diagramas de interação.

O diagrama de visão geral tem a mesma estrutura do diagrama de atividades, trocando as atividades por diagramas de sequência que mostram as classes, além de mensagens envolvidas em cada caso de uso.

Principais componentes: diagramas de sequência, decisões, fluxos.

Como utiliza os mesmos blocos de construção do diagrama de atividades, podemos verificar, passo a passo, as interações das classes em cada uma das sequências criadas no diagrama de visão geral de interação.

Vejamos um exemplo desse diagrama, modelando a sequência de atividades desde o registro até o pagamento da compra (figura 128).

Podemos observar passo a passo o registro de produtos comprados pelo cliente, tanto por meio do computador quanto da balança, analisando cada interação até o pagamento, que finaliza o processo.

4.4. Exemplo de desenvolvimento de projetos utilizando UML

Geralmente, desenvolvemos os diagramas de casos de uso para agrupar as funcionalidades mais importantes a serem implementadas. Sobre tais funcionalidades criamos o diagrama de classes, que será a estrutura de nossa aplicação ou o que chamamos de classes de projeto. No caso da padaria do senhor João, as classes de projeto são as classes de cliente, produto, funcionário, compra, fornece e suas classes filhas.

Definimos seus principais atributos e então fazemos o diagrama de sequência para os casos de uso mais relevantes – no exemplo da padaria, os casos de uso registrar compra, pagar compra, receber mercadoria. Usamos esse diagrama para definir os principais métodos de nossas classes e as trocas de mensagens entre elas. Com isso definido, voltamos ao diagrama de classes e o complementamos com os métodos definidos nos diagramas de sequência.

sd registra_Compra

:Funcionario | FTelaRegistraProduto | Funcionario | Cliente | Produto

loop [funcOK==false]

1:login()

1.1:funcOK=verificaFuncionario (String:cpf,String:senha):boolean

loop [clienteOK==false]

2:leCartao()

2.1:clienteOK=verificaCartao (Cartao:cartao):boolean

loop [enquantohouverproduto]

loop[valorUnitario==0]

3:leProduto()

3.1:valorUnitario=verificaProduto (Produto:produto,double:quantidade):double

4:confirmaCompra()

<<create>>

4.1:ItemCompra(Produto:produto,double:quantidade, double:precoUnitarioVenda,Funcionario:funcionario)

ItemCompra

sd registra_Pagamento

:Caixa | :FRegistraPagamento | :Cliente | :ItemCompra

1:leCartao ()

loop [CartaoOK==false]

1.1:CartaoOK=IdentificaCartao (String:nroCartao):boolean

2:itens[] = totalizaCompra (String:nroCartao):ItemCompra[]

(nroCompra is null and dataPagamento is null)

3:selecionaItensCompra

4:exibeTotalCompra()

5:registraPagamento (double:valorPago,String:formaPagto)

5.1:calculaTroco()

5.1.1:abreGaveta()

(formaPagto='cartao')

5.1.1.1:conectaOperadoraDeCartao (double:valorPago) : double

5.2:atualizaItemCompra (data:DataPagamento,integer:nroCompra)

5.3:Compra(date:Data,double:totalCompra, Caixa:caixa,String:formaPagto)

:Compra

sd registra_Compra_Balanca

:Atendente | Balanca | :Funcionario | :Produto | :Cliente

loop [funcionarioOK==false]

1:login()

1.1:funcionarioOK=verificaFuncionario (String: CPF,String:senha):boolean

2:calculaPesoProduto

loop [precoPorQuilo=0]

3:leProduto()

3.1:precoPorQuilo = verificaProduto (String:codProduto) : double

4:calculaValorTotalItem()

loop [clienteOK==false]

5:leCliente()

5.1:clienteOK=verificaCliente(String:nroCartao):boolean

6:confirmaCompra()

<<create>>

6.1:ItemCompra(Funcionario:func, Cliente:cart,Produto: prod,double:quantidade,double:valorTotal)

:ItemCompra

Figura 128
Diagrama de visão geral de interação.

Começamos então a nos preocupar com a interface e com o usuário e escolhemos quais formulários serão necessários para o funcionamento de nossa aplicação.

Com as classes definidas, começamos a separá-las para melhor organizar a aplicação. Utilizamos para isso o padrão MVC. Esse padrão, muito utilizado pelos desenvolvedores orientados a objeto, foi criado com a linguagem de programação Smalltalk80. Consiste basicamente em dividir as classes da aplicação em três grupos, que chamamos de model, view e controller. Criamos um pacote para cada um desses itens. No model, em geral, inserimos as classes de projeto, seguindo o modelo de nossa aplicação. No view criamos as classes de interface com o usuário (telas) e no controller inserimos as classes que implementarão as funcionalidades de cada classe, desde uma simples consistência até o acesso ao banco de dados. Essas classes que implementam o acesso ao banco de dados levam o nome de classes de persistência – muitos desenvolvedores as colocam em outro pacote.

Dentro do view voltamos a separar as classes em pacotes para implementar a modulação do sistema, criando pacotes para cadastro, movimentação, consultas, relatórios e demais rotinas da execução do sistema, às quais chamamos de ferramentas. Com isso organizamos internamente as classes nos mesmos padrões utilizados na aplicação.

DICA
Você pode pesquisar exemplos de implementação a partir de MVC em livros e sites sobre linguagens de programação. Consulte especificamente bibliografias que abordem a linguagem Java.

Terminada a separação das classes, criamos os diagramas de atividade para os casos de uso cujo procedimento e funcionamento pretendemos deixar documentados.

Verificamos então se é necessário documentar as interfaces de algum componente de software e elaboramos diagramas de componentes para isso.

Se houver classes que mudam de estado no decorrer da execução do sistema, desenvolvemos o diagrama de máquinas de estados para demonstrar qual a ideia da mudança de estado para cada uma delas.

Por fim, se o ambiente de implantação for um ambiente heterogêneo, isto é, que envolve arquitetura com vários servidores, como servidor de banco de dados e servidor de aplicações, entre outros, criamos o diagrama de implantação para demonstrar a arquitetura de software e hardware onde a aplicação deverá ser instalada.

Veja que nessa forma de desenvolvimento utilizamos os diagramas de casos de uso, os de classes, os de sequência, os de pacotes, os de atividades, o de máquina de estados, os de componentes e o de implantação.

Pesquise sobre padrões de projeto (design patterns). São 24 padrões de desenvolvimento de software orientado a objetos que se propõem a resolver os problemas mais comuns nesse processo.

Tudo depende do tamanho da aplicação a ser desenvolvida e das dificuldades que encontramos nas fases de análise e projeto de software.

É comum também surgirem alterações nos modelos na fase de programação do software. Nesse caso, voltamos ao modelo e incluímos as alterações que fizemos na fase de programação e testes. Mesmo depois de implantada a solução, sempre que for necessário fazer alguma alteração no sistema, devemos voltar ao modelo e fazer a inclusão, para que o modelo nunca fique diferente do **sistema criado**.

Considerações finais

Longe da tentativa de esgotar os assuntos aqui abordados, a intenção deste livro é ajudá-lo compreender um pouco melhor as fases de um projeto, o Modelo Relacional, o Modelo de Entidade e Relacionamento, o SGBD, o método orientação a objetos, o SQL e a UML.

Se você escolheu a área de informática para atuar profissionalmente, continue estudando e aprendendo sempre, pois nesse campo, dinâmico, as mudanças são constantes e quem não se atualiza vai ficando para trás. Esperamos que você tenha conseguido alcançar uma boa visão sobre os temas aqui abordados e que vá agora em busca de mais informações para aprofundar seus conhecimentos para avançar cada vez mais na sua carreira profissional.

Referências bibliográficas

AUGUST, Judy H. *JAD - Joint Application Design*. São Paulo: Makron Books, 1993.

BEZERRA, E. *Princípios de análise e projetos de sistema com UML*. Elsevier, 2007.

BOOCH, G., RUMBAUGH, J. e JACOBSON, I. – *UML Guia do Usuário*. 2ª edição. Elsevier, 2005.

COSTA, R. L. C. , *SQL Guia Prático*, 2ª edição. Rio de Janeiro: Brasport, 2006.

DA ROCHA, Ana Regina Cavalcanti et al. (org.). *Qualidade de software: Teoria e prática*. São Paulo: Pearson, 2004.

DALTON, Patrick. *Microsoft SQL Server Black Book*. 5ª edição. The Coriolis Group, 2008.

DATE, C. J. *Introdução a Sistemas de Banco de Dados*. 7ª edição. Rio de Janeiro: Campus, 2000.

ELMASRI, S. N.; NAVATHE, B. S. *Sistemas de Banco de Dados*: Fundamentos e Aplicações. 3ª edição. Rio de Janeiro: Livros Técnicos e Científicos, 2002.

ELMASRI, S. N.; NAVATHE, B. S. *Sistemas de Banco de Dados*. 4ª edição. São Paulo: Pearson Education, 2005.

FOWLER, Chad. *The Passionate Programmer*: Creating a Remarkable Career in Software Development, 2009.

KORTH, Henry F. e SILBERSCHATZ. *Sistemas de Bancos de Dados*, Ed. Mc.Graw-Hill, SP, 2ª edição revisada,1995.

MACHADO, F. N. R.; ABREU, M. *Projeto de Banco de Dados*, Uma Visão Prática. 2ª edição. São Paulo: Editora Érica, 1996.

OLIVEIRA, J. Wilson. *Oracle 8i & PL/SQL*. 1ª edição. Santa Catarina: Editora Visual Books, 2000.

OLIVEIRA, J. Wilson. *SQL Server 7 com Delphi*. 1ª edição. Santa Catarina: Editora Visual Books, 2001.

ORIT, Dubinsky Yael Hazzan. *Agile Software Engineering*. 1ª edição. Springer, 2008.

Project Management Body of Knowledge (PmBok). 3ª edição, 2004.

PRESSMAN, Roger S. *Engenharia de Software*. São Paulo: Pearson, 2006.

SILBERSCHATZ, Abraham; KORTH, H. F. ; SUDARSHAN, S. *Sistema de Banco de Dados*. 3ª edição. São Paulo: Makron Books, 1999.

SOMMERVILLE, Ian. *Engenharia de Software*. São Paulo: Pearson, 2004.

SWEBOK, *Software Engineering Body of Knowledge*, 2004.

LARMAN, C. *Utilizando UML e Padrões*. 3ª edição. Bookman, 2007.

WELLING, L. THOMSON L, *Tutorial MySQL*. 1ª edição. Rio de Janeiro: Editora Ciência Moderna, 2003.

YOURDON, Edward. *Declínio e Queda dos Analistas e Programadores*. São Paulo: Makron Books, 1995.

Excelência no ensino profissional

Administrador da maior rede estadual de educação profissional do país, o Centro Paula Souza tem papel de destaque entre as estratégias do Governo de São Paulo para promover o desenvolvimento econômico e a inclusão social no Estado, na medida em que capta as demandas das diferentes regiões paulistas. Suas Escolas Técnicas (Etecs) e Faculdades de Tecnologia (Fatecs) formam profissionais capacitados para atuar na gestão ou na linha de frente de operações nos diversos segmentos da economia.

Um indicador dessa competência é o índice de inserção dos profissionais no mercado de trabalho. Oito entre dez alunos formados pelas Etecs e Fatecs estão empregados um ano após concluírem o curso. Além da excelência, a instituição mantém o compromisso permanente de democratizar a educação gratuita e de qualidade. O Sistema de Pontuação Acrescida beneficia candidatos afrodescendentes e oriundos da Rede Pública. Mais de 70% dos aprovados nos processos seletivos das Etecs e Fatecs vêm do ensino público.

O Centro Paula Souza atua também na qualificação e requalificação de trabalhadores, por meio do Programa de Formação Inicial e Educação Continuada. E ainda oferece o Programa de Mestrado em Tecnologia, recomendado pela Capes e reconhecido pelo MEC, que tem como área de concentração a inovação tecnológica e o desenvolvimento sustentável.

Fundação
Padre Anchieta